Cranfield University

Cranfield Defence and Security

Department of Informatics & Systems Engineering
Applied Mathematics & Scientific Computing

PhD Thesis

Academic Year 2011 - 2012

Rahul Vijay Kharche

MATLAB Automatic Differentiation using Source Transformation

Supervisor: Dr. S.A. Forth

August 2011

Abstract

This thesis presents our work on compiler techniques to implement Algorithmic Differentiation (AD) using source transformation in MATLAB. AD is concerned with the accurate and efficient computation of derivatives of complicated mathematical functions represented by computer programs.

Source transformation techniques for AD, whilst complicated to implement, are known to yield derivative code with better run-time efficiency than methods using overloading support of the underlying language. We present results from MSAD that confirm the increase in efficiency using source transformed code for MATLAB AD. Most importantly, we demonstrate the use of a unique compiler code specialisation method to implement AD. We also assert the need for compiler optimisations in MATLAB, especially in the context of AD, and showcase MSAD as an extensible infrastructure to implement new optimisations and algorithms for AD or other applications.

Where other efforts on MATLAB AD are implemented using operator overloading or a mix of overloading and source transformation, MSAD (Springer LNCS, Vol. 3994, 2006) was the first to generate differentiated MATLAB code using source transformation alone. MSAD is also the only effort to implement source transformed AD by resolving overloaded MATLAB code. The existing MAD package (ACM TOMS, **32**, No.2, 2006) provides a highly efficient overloaded implementation of MATLAB AD. MSAD uses compiler code specialisation techniques to specialise and inline `fmad` and `derivvec` overloaded operations of the MAD package in order to generate MATLAB AD code. The operator overloading overheads inherent in MAD are eliminated while preserving the `derivvec` class's optimised derivative combination operations.

As a compiler framework for MATLAB, MSAD demonstrates a novel use of an existing effective compiler algorithm (Sparse Conditional Constant Propagation) to infer properties of MATLAB variables such as type, rank,

shape, sparsity and value by propagating a composite lattice of all the properties together.

कर्मण्येवाधिकारस्ते मा फलेषु कदाचन। मा कर्मफलहेतुर्भूर्मा ते सङ्गोऽस्त्वकर्मणि ॥

श्रीमद्भगवद्गीता ॥ २ - ४७ ॥

You have a right to perform your prescribed duty, but you are not entitled to the fruits of action. Never consider yourself the cause of the result of your activities, and never be attached to not doing your duty.

Bhagavad Gita, 2:47

*Dedicated to my father who taught me the meaning of **karma**, and whose passion for science and philosophy alike, is a source of inspiration. And to my mother whose gentle, yet loving concern has been a strong moral support.*

Acknowledgements

First and foremost, I would like to thank my supervisor Dr. Shaun Forth, who has been very supportive and patient throughout the length of my part-time research effort, and always encouraged me to think independently. I would also like to thank Shaun's lovely family who have always been very warm and welcoming since I have known them.

I would like to thank both Dr. Venkat Sastry and Dr. Shaun Forth for the opportunity to work at Cranfield as a Research Officer, which made this part-time research possible.

I would also like to thank my present employer NVIDIA (previously Icera Inc.), particularly my line-managers Alan Alexander and Dave Edwards who have been very understanding of my part-time research commitments.

Very importantly, I would like to extend a heart-felt thanks to my family and friends who have motivated and selflessly supported this endeavour through the years.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Why derivatives?

The availability of ever larger computing power fuels the demand for more complex and accurate models of physical systems, which in turn increases the complexity of the numerical solvers involved. Among the most common scientific computing sub-problems solved as a part of these models, are nonlinear optimisations and solutions to nonlinear equations. Implicit numerical methods applied to solve stiff ODEs, and shooting methods in boundary value problems also result in a system of nonlinear equations [HW76]. Numerical schemes that rely on derivatives of the sub-problem are very often employed to solve them. For example, Newton's method for solving a system of nonlinear equations uses the **Jacobian** of the system of equations, and Newton iterations in Large-Scale optimisation use the **gradient** and **Hessian** of the objective function [NW99].

In large optimisation problems the objective functions often tend to have sparse second order derivatives, or the functions themselves are *partially separable* [GW08, Ch.11]. Also nonlinear equations may have a sparse Jacobian, especially if they are based on a discretisation with compact stencil. In most practical cases the number of variables involved in such sub-problems are in the order of several thousands. In order to obtain the solution of the complete problem in a reasonable time, it becomes vital to exploit any underlying

properties of derivatives or function to compute derivatives efficiently. Advanced techniques like Jacobian compression [GW08, Ch.8] to compress the domain or range of the problem based on the sparsity pattern or partial separability, dynamic propagation of sparse derivatives or static techniques such as vertex elimination [GW08, Ch.9] are employed to increase the efficiency of derivative computation, and hence the numerical solvers. Furthermore, the convergence characteristics of iterative numerical solvers often depend closely on the accuracy of the derivatives. For example, in gradient based optimisation solvers, the gradients of the objective and constraints form a part of the *Karush Kuhn Tucker* conditions used directly to solve the problem [NW99]. This implies that the inaccuracies in evaluating the KKT conditions limit the accuracy of the solution [Gri93]. Thus, not only is there a need to compute derivatives efficiently, but also for them to be sufficiently accurate.

1.2 Computing derivatives

The first obvious choice, is to compute derivatives of the required function *by hand*. Differentiation is carried out by systematically applying the *chain rule* for differentiation. The computed derivatives will be exact owing to the use of the chain rule. Since most solvers are implemented as software libraries or routines inside software packages, it is required to code the differentiated form in a programming language like C, C++, Fortran or MATLAB[1]. While translating to a programming language, the expressions can also be hand optimised to make the derivative computation efficient. The difficulty with this approach is that as problems grow large, the complexity of computing derivatives, optimising the expressions and coding them, may become increasingly difficult to handle. The process is also inherently error prone.

An alternative is to make use of *symbolic manipulation* packages like Maple or Mathematica. When supplied with the algebraic specification for the function, the symbolic manipulations tools produce a new set of algebraic expressions for the derivatives. As in the previous case, these derivative

[1]MATLAB is a trademark of The MathWorks, Inc.

expressions are then translated to a programming language and provided to the solver routine. The problem with this approach is that the generated expressions may be long and unwieldy, and possibly in a form that will not execute efficiently.

A common limitation of the earlier two approaches is that a function cannot harness more expressive constructs like loops, branches or sub-functions of a programming language. It would therefore be convenient to translate the original function definition into a programming language, and have it differentiated automatically.

1.2.1 Divided Differencing

The method of divided differences, commonly used in approximating the derivatives of a function, permits the differentiation of user programs numerically, without the user having to delve into the function intrinsics. The derivative of a function with respect to any of its component may be approximated using one-sided, central or higher order differences. *Divided differencing* or *finite differencing* methods are based on Taylor's theorem. According to Taylor's theorem, if a function $f : \mathbb{R}^n \to \mathbb{R}$ is twice continuously differentiable and $\mathbf{x}, \mathbf{p} \in \mathbb{R}^n$, then

$$f(\mathbf{x} + \mathbf{p}) = f(\mathbf{x}) + \nabla f(\mathbf{x})^T \mathbf{p} + \frac{1}{2}\mathbf{p}^T \nabla^2 f(\mathbf{x} + t\mathbf{p})\mathbf{p}, \tag{1.1}$$

for some $t \in (0, 1)$. The term ∇f is the gradient of the function f, and $\nabla^2 f$ the Hessian. If $||\nabla^2 f(\cdot)||$ is bounded by L in a small region around x and we let $\mathbf{p} = \epsilon \mathbf{e}_i$, to be the perturbation along the ith component of the input vector $\mathbf{x}$, then according to [NW99, p.167]

$$\frac{\partial f}{\partial x_i}(\mathbf{x}) = \frac{f(\mathbf{x} + \epsilon \mathbf{e}_i) - f(\mathbf{x})}{\epsilon} + \delta_\epsilon \quad \text{where,} \;\; |\delta_\epsilon| \leq (L/2)\epsilon. \tag{1.2}$$

For small values of the perturbation ϵ, the error term δ_ϵ in (1.2) can be ignored and we get the *forward difference* approximation (1.3) to the ith

component of the gradient.

$$\frac{\partial f}{\partial x_i}(\mathbf{x}) \approx \frac{f(\mathbf{x} + \epsilon \mathbf{e}_i) - f(\mathbf{x})}{\epsilon} \tag{1.3}$$

Computer implementations that use floating point arithmetic to compute the value of the function f involve *roundoff errors*. These are relative errors involved with each floating point operation performed, and are individually bounded by $\mathtt{u}$, the *unit roundoff error*. If L_f is a bound on $||f(\cdot)||$, the norm of the function value, then the bound on the absolute error involved in computing the function value is of the order of $L_f\mathtt{u}$. The bound on the roundoff error in the complete operation in (1.3) combined with the bound on the truncation error term in (1.2) gives the bound on the total error in computing the derivative using the forward difference approximation. If this total error is minimised with respect to the perturbation ϵ, we find a near optimal value to be $\epsilon = \sqrt{\mathtt{u}}$. Nocedal and Wright [NW99, p.168] further deduce that the total error in computing the derivative is close to $\sqrt{\mathtt{u}}$. We thus see that the maximum accuracy obtainable using the forward difference approximation, is at most half the number of digits in the working precision. The computational cost involved in this method is one additional function evaluation for each component of the input. A further few digits of accuracy may be achieved by using the central difference formula

$$\frac{\partial f}{\partial x_i}(\mathbf{x}) \approx \frac{f(\mathbf{x} + \epsilon \mathbf{e}_i) - f(\mathbf{x} - \epsilon \mathbf{e}_i)}{2\epsilon},$$

but at the cost of a further function evaluation for each component of the input. The optimum perturbation size in this case is $\epsilon = \mathtt{u}^{2/3}$.

The divided difference method can also be extended to computing the Jacobian of a vector function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$. The Jacobian $J(x)$ is computed one column at a time, setting $\mathbf{p} = \epsilon \mathbf{e}_i$. The entire Jacobian is computed in $n + 1$ function evaluations. The ith column is then given by the expression

$$\frac{\partial \mathbf{f}}{\partial x_i}(\mathbf{x}) \approx \frac{\mathbf{f}(\mathbf{x} + \epsilon \mathbf{e}_i) - \mathbf{f}(\mathbf{x})}{\epsilon} \tag{1.4}$$

The `numjac` [SR97] routine of MATLAB uses the forward difference method (1.4) with adaptive perturbation sizes ϵ, and returns the complete Jacobian of a function. Each column of the Jacobian is calculated with a common perturbation size set such that the largest entry in a column is calculated accurately. These sizes are also adjusted across calls to the `numjac` routine. However, this method of Jacobian calculation can be prone to errors as the smaller entries can have significant relative error, as shown by the example in Shampine, Ketzscher and Forth [SKF05].

The method of divided differences is prone to truncation and significance errors reducing the accuracy of computed derivatives by several orders of magnitude over machine precision. The time complexity of gradient computation is p times the time required to compute the original function, where p is the number of directional derivatives computed. Compression techniques can be applied to compute derivatives efficiently where derivatives are sparse and the sparsity structure is known before-hand [GW08, Ch.7].

1.2.2 Automatic Differentiation

Automatic or *Algorithmic Differentiation* (AD) [GW08] is concerned with the accurate and efficient evaluation of derivatives for functions defined by computer programs. Automatic differentiation techniques work on the premise that any program definition of a function, no matter how complicated, can be expressed as a finite set of *elementary operations* e.g. additions, multiplications, trignometric functions. If we assume that these elementary operations are differentiable, then the *chain rule* of differentiation can be applied systematically to each elementary operation to finally give the derivatives of the complete function with respect to the independent variables. The use of the chain rule makes the method free from truncation and significance errors, and the derivatives are computed correct to machine precision.

A function $\mathbf{y} = \mathbf{f}(\mathbf{x})$, with $\mathbf{y} \in \mathbb{R}^m$ and $\mathbf{x} \in \mathbb{R}^n$, in the form of a computer program is therefore viewed as a sequence of elementary binary or unary operations, or calls to other sub-functions. The result of each such elementary operation is assigned to an intermediate variable and further references to this

operation are replaced by the intermediate variable. For further analysis, it becomes convenient to represent all the variables in the program using indices; we therefore label all the variables (independent, intermediate and dependent) as v_i. The indexing convention used to partition these into the independent ($\mathbf{x}$), intermediate ($\mathbf{z}$), and dependent ($\mathbf{y}$) variables is:

$$\mathbf{x} = v_{1-n}, v_{2-n}, \cdots, v_0 \tag{1.5}$$

$$\mathbf{z} = v_1, v_2, \cdots, v_{l-m} \tag{1.6}$$

$$\mathbf{y} = v_{l-m+1}, v_{l-m+2}, \cdots, v_l \tag{1.7}$$

The relationship between these variables is then summarised by,

$$v_i = \varphi_i(v_j)_{j \prec i}, \quad \text{for} \quad i > 0 \tag{1.8}$$

where φ_i represents elemental functions or elementary operations, and the precedence operator $j \prec i$ indicates that v_j is required to compute v_i with $j < i$. With this convention the entire function definition can be expressed in the form of elementary operations as given by Table 1.1. This form is called an *evaluation procedure* [GW08, Ch.3].

v_{i-n}	$=$	x_i	$i = 1, \cdots, n$
v_i	$=$	$\varphi_i(v_j)_{j \prec i}$	$i = 1, \cdots, l$
y_{m-i}	$=$	v_{l-i}	$i = m-1, \cdots, 0$

Table 1.1: Evaluation procedure

For example consider the sample function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ with $\mathbf{y} \in \mathbb{R}^2$ and $\mathbf{x} \in \mathbb{R}^3$ given in Table 1.2 [FTPR04]. An evaluation procedure for this function is given by the left column in Table 1.3.

The classical AD process then reduces to selecting a method of propagating the derivatives and applying the chain rule of differentiation to each elementary operation $\varphi_i(\cdot)$. There are two primary methods for propagating the derivatives, forward and reverse. Since the implementation described in this thesis uses the forward mode, it is considered here in detail.

$$
\begin{aligned}
\text{function}[y_1, y_2] &= \text{func}(x_1, x_2, x_3) \\
w_1 &= \log(x_1 * x_2) \\
w_2 &= x_2 * x_3^2 - a \\
w_3 &= b * w_1 + x_2 / x_3 \\
y_1 &= w_1^2 + w_2 - x_2 \\
y_2 &= \sqrt{w_3} - w_2
\end{aligned}
$$

Table 1.2: Sample program

Forward mode

v_{-2}	$=$	x_1	$\dot{v}_{-2}$	$=$	$\dot{x}_1$
v_{-1}	$=$	x_2	$\dot{v}_{-1}$	$=$	$\dot{x}_2$
v_0	$=$	x_3	$\dot{v}_0$	$=$	$\dot{x}_3$
v_1	$=$	$v_{-2} * v_{-1}$	$\dot{v}_1$	$=$	$\dot{v}_{-2} * v_{-1} + v_{-2} * \dot{v}_{-1}$
$v_2 = w_1$	$=$	$\log(v_1)$	$\dot{v}_2$	$=$	$\dot{v}_1 * (1/v_1)$
v_3	$=$	v_0^2	$\dot{v}_3$	$=$	$2v_0 * \dot{v}_0$
v_4	$=$	$v_{-1} * v_3$	$\dot{v}_4$	$=$	$\dot{v}_{-1} * v_3 + v_{-1} * \dot{v}_3$
$v_5 = w_2$	$=$	$v_4 - a$	$\dot{v}_5$	$=$	$\dot{v}_4$
v_6	$=$	$1/v_0$	$\dot{v}_6$	$=$	$-(v_6 * v_6) * \dot{v}_0$
v_7	$=$	$v_{-1} * v_6$	$\dot{v}_7$	$=$	$\dot{v}_{-1} * v_6 + v_{-1} * \dot{v}_6$
v_8	$=$	$b * v_2$	$\dot{v}_8$	$=$	$\dot{v}_2$
$v_9 = w_3$	$=$	$v_8 + v_7$	$\dot{v}_9$	$=$	$\dot{v}_8 + \dot{v}_7$
v_{10}	$=$	v_2^2	$\dot{v}_{10}$	$=$	$2v_2 * \dot{v}_2$
v_{11}	$=$	$v_{10} + v_5$	$\dot{v}_{11}$	$=$	$\dot{v}_{10} + \dot{v}_5$
v_{12}	$=$	$\sqrt{v_9}$	$\dot{v}_{12}$	$=$	$0.5 * \dot{v}_9 / v_{12}$
v_{13}	$=$	$v_{11} - v_{-1}$	$\dot{v}_{13}$	$=$	$\dot{v}_{11} - \dot{v}_{-1}$
v_{14}	$=$	$v_{12} - v_5$	$\dot{v}_{14}$	$=$	$\dot{v}_{12} - \dot{v}_5$
y_1	$=$	v_{13}	$\dot{y}_1$	$=$	$\dot{v}_{13}$
y_2	$=$	v_{14}	$\dot{y}_2$	$=$	$\dot{v}_{14}$

Table 1.3: Evaluation procedure and tangents for function in Table 1.2

If we supply the directional derivatives $\dot{x}_i$ corresponding to the independents x_i, the the forward mode can be seen as applying the chain rule of differentiation to each statement in Table 1.1 successively as is shown in Table 1.4. The elemental functions φ_i are assumed to be differentiable. Forward mode AD applied to the sample program in Table 1.2 is shown in Table 1.3.

$\dot{v}_{i-n}$	$=$	$\dot{x}_i$	$i = 1, \cdots, n$
v_{i-n}	$=$	x_i	
$\dot{v}_i$	$=$	$\sum_{j \prec i} \frac{\partial}{\partial v_j} \varphi_i(v_j) \dot{v}_j$	$i = 1, \cdots, l$
v_i	$=$	$\varphi_i(v_j)_{j \prec i}$	
$\dot{y}_{m-i}$	$=$	$\dot{v}_{l-i}$	$i = m-1, \cdots, 0$
y_{m-i}	$=$	v_{l-i}	

Table 1.4: Forward mode evaluation procedure

During execution, the derivative computations, shown in the right column, are interspersed with the computations of the elemental functions on the left. Typically, the derivatives $\dot{v}_i$ are computed in lock step, just before the corresponding variable v_i

The sequence of operations in Table 1.1 can equivalently be represented as an extended system of nonlinear equations

$$E(x; v) \equiv (\varphi(u_i) - v_i)_{i=1-n,\cdots,l} \; = \; 0, \tag{1.9}$$

where $u_i \equiv (v_j)_{j \prec i}$. For $i < 0$, $\varphi_i(u_i)$ is simply equal to x_{i+n}. If we differentiate this system of nonlinear equations, we get

$$E'(x; v) = \left(\frac{\partial \varphi_i(u_i)}{\partial v_j} - \delta_{ij} \right), \tag{1.10}$$

where $1 - n \leq i \leq l$, $1 \leq j \leq i$, and δ_{ij} is the Kronecker delta function. Due to the precedence relation on the elemental functions in (1.8), the Jacobian matrix $E'(x; v)$ is a unitary lower triangular matrix [GW08, Ch.3]. Then according to the forward mode, derivatives of the function F associated with the extended system $E(x; v)$ can also be obtained by solving the following linear system

$$E'(x; v) \nabla V = R \tag{1.11}$$

where ∇V is a vector of derivatives $\dot{v}_j$, and the first n rows of the $(l + n) \times n$ matrix R select the directions along which the derivatives are determined (all

other entries in the matrix R being zero).

Reverse mode

Reverse mode of AD uses a backward application of the chain rule in order to compute the derivatives of a function. The method propagates *adjoints* or sensitivities of each variable with respect to the dependents in a reverse sweep through a program. For a vector function $\mathbf{y} = \mathbf{F}(\mathbf{x})$, with $\mathbf{y} \in \mathbb{R}^m$ and $\mathbf{x} \in \mathbb{R}^n$, the adjoint operations are defined as:

$$\overline{x}_i \equiv \frac{\partial F}{\partial x_i} \quad \text{and} \quad \overline{y}_j \equiv \frac{\partial F}{\partial y_j} \tag{1.12}$$

The reverse mode evaluation procedure for the program in Table 1.1 is given in Table 1.5.

v_{i-n}	$=$	x_i	$i = 1 \cdots n$
v_i	$=$	$\varphi_i(v_j)_{j \prec i}$	$i = 1 \cdots l$
y_{m-i}	$=$	v_{l-i}	$i = m-1 \cdots 0$
$\overline{v}_{l-i}$	$=$	$\overline{y}_{m-i}$	$i = 0 \cdots m-1$
$\overline{v}_j$	$=$	$\sum_{i \succ j} \overline{v}_i \frac{\partial}{\partial v_j} \varphi_i(u_i)$	$j = l-m \cdots 1-n$
$\overline{x}_i$	$=$	$\overline{v}_{i-n}$	$i = n \cdots 1$

Table 1.5: Reverse mode evaluation procedure

From the program description we observe that the reverse mode operates in two phases: one performing a forward sweep through the program computing the values of v_i, while saving some that are required in the following phase; and the second a reverse sweep that propagates the sensitivities of the dependent variables with respect to the intermediate or independent under consideration. The result of the entire operation is the adjoint $\overline{x} = \overline{y}\ F'(x)$.

The Jacobian of the extended system in (1.9) can be used to to compute the adjoints by solving the linear system

$$E'(x;v)^T \overline{A} = P, \tag{1.13}$$

where $\overline{A}$ is the adjoint vector and the last m rows of the $(n+l) \times m$ matrix

P gives the direction (all other entries in the matrix P being zero).

Assuming differentiability of elementary functions, AD provides derivatives accurate to machine precision. AD also provides a priori bounds on the cost of computing derivatives in terms of the cost of evaluating the input function. However, determining the exact time complexity of derivative computation is limited by modern computer architectures with complex memory subsystems, multi-processor or superscalar CPUs with SIMD or out-of-order execution. The following time complexity measure assumes a simplistic model that accounts only for the number of basic arithmetic operations and memory moves. Tangent propagation in the forward mode of AD has a time complexity $\omega_{tangp}\, CPU(f)$, where $CPU(f)$ is the time required to compute the function value, $\omega_{tangp} \in [1+p, 1+1.5p]$, and p, is the number of tangents being propagated, usually the total number of independents to the function [GW08, p.80]. Gradient computation though reverse mode of AD has a time complexity $\omega_{gradq'}\, CPU(f)$, where $\omega_{gradq'} \in [1+2q, 1+2.5q]$, and q is the number of gradients being propagated, usually the total number of dependents output from the function [GW08, p.83]. The forward mode of AD is efficient in computing derivatives when the number of independents (that we wish to differentiate with respect to) is small. The reverse mode AD is more efficient if the number of dependents is known to be small. Like in divided differencing, to compute sparse derivatives efficiently, compression techniques can be applied if the sparsity structure is known in advance. If the sparsity structure is unknown, with AD it is possible to dynamically exploit derivative sparsity by propagating sparse derivatives and maintain efficiency in derivative computation [GW08, Ch.7].

The motivation to use AD generated derivatives in place of the conventional finite-differencing, is to improve solution quality and speed. As mentioned in Section 1.1, this improvement becomes possible because the convergence characteristics of iterative numerical solvers depend on the accuracy of the computed derivatives. Where finite-differencing suffers from truncation and significance errors, the recursive use of chain rule of differentiation in AD gurantess derivatives correct to working precision.

1.3 Automatic Differentiation Tools

To complement developments in AD on the algorithmic front, the past decade or two has seen significant developments in tools that apply AD methods to computer programs in languages native to scientific computing. AD tools have been successfully used in large scientific computing problems like the Adjoint MIT Ocean General Circulation Model [HHG05], Sensitivity Analysis of Mesoscale Weather Model [BPK96], Reducing Sonic Boom under a Supersonic Jet [HVD03], Satellite Orbit Simulation [KWBV05], Satellite Boom Design [TFK05] and several others.

As the choice of programming language for numerical computing varies between Fortran, C/C++ and MATLAB, AD tools are built for the corresponding languages. Some of these include ADIFOR [BCC+92], TAF [Fas03] and TAPENADE [HGP03] for use with Fortran, ADIC [BRM97], ADOL–C [GJU96] and COSY INFINITY [BMS+96] for C/C++ and ADMAT [CV98b], MAD [For06] and ADiMat [BBL+02] for MATLAB.

These tools make use of underlying language features or compiler-like algorithms to effect the differentiation of the input code. Although the underlying principle in computing the derivatives using AD is the same, there are two different implementation methods, *overloading* and *source transformation*.

Overloading

In programming languages all intrinsic operators like `'+'`, `'*'`, etc. and library operations such as `'sin'`, `'exp'`, etc. have predefined semantics that determine how the operator or operation operates on its operand(s). Object oriented paradigms (OOP) support *overloading* that allows the developer to supply new data types together with programmed procedures to modify the semantics of the operations or operators that operate on them. For example, the intrinsic data types such as `double` or `int` can be overloaded by a custom class, say `ddouble` that stores value of the variable and its derivatives. If variables `'a, b, y'` are declared to be of type `ddouble` then the overloaded `'+'` operator in the operation `'y = a + b'` would com-

pute the derivative of `'y'` as `'y.der = a.der + b.der'` together with the value `'y.val = a.val + b.val'`. Here, the task of resolving the right operation or operator, depending on the type of its operands, is delegated to the compiler or even an interpreter in a runtime environment. Languages like Fortran (Fortran 90 onwards), C++ and MATLAB support overloading. And tools such as ADMAT [CV98a], MAD [For06], ADOL–C [GJU96] and COSY INFINITY [BMS+96] use this overloading feature to implement AD.

Source Transformation

The source transformation approach makes use of more complex source code analysis and transformations to produce a similar effect to that achieved by overloading - a compiler-like tool processes the input code prior to execution [ASU86], and generates a new program that on execution computes the desired derivatives. Where overloading defers the resolution of operators and operations until compile time or even execution time, source transformation systematically breaks down the input program into individual operations and operators and inserts the relevant derivative procedure or set of derivative operations directly into the generated source. The previous example `'y = a + b'` would result in the operation `'der_y = der_a + der_b'` being inserted into the generated source along with the original operation, when using the forward mode of AD. In the case of the reverse mode, during the reverse sweep adoints operations `'adj_a += adj_y'`, `'adj_b += adj_y'` and `'adj_y = 0'` are inserted instead. ADIC [BRM97], TAPENADE [HGP03], TAF [Fas03], ADIFOR [BCC+92] and ADiMat [BBL+02] make use of source transformation to implement AD.

1.3.1 Compiler Support for AD tools

The process of differentiating programs, as described in Section 1.2.2, is carried out by augmenting an input program with the corresponding derivative code. This can be done using operator overloading or source transformation. While augmenting the program, care is taken to reuse any computed values to avoid redundant computations. However, the scope of reuse generally does

not extend beyond a unary or binary operation, causing re-computations of sub-expressions or even expressions across statements. This also applies to copying of variable values and algebraic simplifications involving constant values. The computational complexity of the underlying problem is scaled by the complexity of the augmented AD operations which accentuates any implementation inefficiencies.

Broadly speaking there are two main models of executing the generated AD code, after a static compilation stage, or dynamically using an interpreter. Typically programs in languages in C, C++, FORTRAN, C# are compiled, whereas MATLAB, Python are interpreted. If the target program gets compiled before execution, as opposed to being interpreted, the compiler can remove any redundant operations thereby increasing code efficiency using rigorous optimisations. The optimizations particularly relevant to AD code, as we will see in the following Chapter 2, are function inlining, common-subexpression elimination, branch optimizations, intra- and inter-procedural constant propagation and constant folding. Compilers also perform machine dependent optimisations on the code to make optimum use of the target processor's instruction set and where possible, the memory subsystem. The subsequent sections discuss the drawbacks of running programs in an interpreted environment, specifically MATLAB.

Programming models for procedural languages encourage the use of modular programming by factoring the program into routines or functions that may be re-used. Often this leads to generalization of function semantics, i.e. a function is written to handle many more cases than those that are frequently executed. Small frequently called functions on the other hand, impose a recurring function call overhead. Function inlining removes the overhead of setting up arguments to a function call, the overhead of the function call itself and increases spatial locality generally making the program more cache friendly. Depending on the static properties of the arguments to a function, inlining can also cause sections of the inlined function to be deemed redundant and removed. Inlining such functions outweighs the function call costs at times. One downside of aggressive inlining is that the code for several small frequently executed functions which can all fit in the cache, may be forced

apart by inlining and compete for cache causing thrashing thereby adversely impacting performance, also documented by Arnold et al. [AFSS00].

In the case of overloaded functions, compile time resolution and inlining also remove the overhead of function signature matching and dispatch. Though operator overloading (ad-hoc polymorphism) may be handled using a *run time dispatch* mechanism [ASU86], an optimising compiler may choose to replace a function call for an overloaded operation with the function body by resolving the class types of overloaded objects. This removes the extra indirection to invoke the function at run-time and can enhance performance further [CG94]. This *inlining* operation may be viewed as an example of source transformation carried out within a compiler.

We believe the performance of programs processed by AD tools rely heavily on the compiler to perform these optimisations. The work presented here highlights code specialization and inlining as the key optimisations to enhance performance of AD operations in MATLAB. The optimization techniques presented here are state of the art compiler techniques that are a part of many industry standard compilers.

1.4 MATLAB Automatic Differentiation

MATLAB, short for *matrix laboratory*, is popular in rapid prototyping and numerical computing owing to its high-level abstraction of matrices that form a fundamental unit of operation, and its rich set of function and GUI libraries. The interpreted nature of MATLAB and its high-level language make programming intuitive and debugging easy. MATLAB also makes use of optimised BLAS and LAPACK routines for internal matrix operations, enabling good performance. MATLAB comes packaged with toolboxes to tackle optimisation, ODE, PDE, control system, neural network and statistical problems, and provisions for data-visualisation, parallelisation and complex model simulations. With a rich environment for numerical computing such as this, the use of AD is clearly befitting to MATLAB based solvers.

1.4.1 AD Tools

The very early efforts at MATLAB AD were by Rich and Hill [HR92] who had a C implementation which could be called from within MATLAB to apply AD to MATLAB codes. The interface into the C implementation was a simple `grad` function that accepted as inputs the MATLAB expression to be differentiated, and the value of the independents at which the expression and its gradients or Jacobian matrix were to be evaluated. The value of the expression and its derivatives at the specified value of the independents were calculated and returned. Although simple, this method was limited to differentiating a MATLAB expression specified by a single string.

ADMAT

Research grade efforts at MATLAB AD only started with ADMAT, the first comprehensive AD tool for MATLAB by Coleman and Verma [CV98a] that uses MATLAB's object-oriented programming features (operator overloading). The ADMAT implementation allows computation of gradients, Jacobian matrices or Hessian matrices. The derivatives can be calculated by both forward and reverse mode of AD. ADMAT also interfaces with ADMIT [CV00], which provides support for computing sparse Jacobian and Hessian matrices via row, column or combined compression techniques. The ADMAT implementation has a large coverage of AD algorithms such as automatic sparsity detection and support for efficient computation of Jacobian-Matrix and Hessian-Matrix products. The problem of optimising the computational complexity is tackled at a high level by exploiting parallelism in the computation. However, not much emphasis has been put on low level effects such as access patterns, spatial locality of derivatives or use of efficient MATLAB program constructs. Also, ADMAT supports only one and two dimensional matrices in both full and sparse computation of derivatives.

ADiMat

ADiMat [BBL$^+$02, Veh01] a MATLAB AD tool developed a little after ADMAT, uses hybrid source transformation and operator overloading approach

to implement forward mode AD. ADiMat also computes gradients and Jacobian matrices using the forward mode of AD. Single directional derivatives are computed completely by augmented MATLAB code alone, however multiple directional derivatives make use of MATLAB overloading feature or alternately use the MATLAB-MEX interfaces to store and compute multiple directional derivatives. Bischof et al. [Veh01] showed that derivatives using forward mode AD of ADiMat were more efficient than that of ADMAT. A Macro language to supply derivatives of user defined functions and MATLAB intrinsic functions to support ADiMat is also defined by Bischof et al. [BBV05]. The ADiMAT tool parses MATLAB using a Bison based LALR(1) grammar into an AST, and performs all code processing and augmentation at the AST level directly.

MAD

The MAD package [For06], developed about the same time as ADiMat, also implements AD for MATLAB using operator overloading. The primary goal of MAD is to make overloaded implementation of AD for MATLAB efficient by optimising the storage of multiple directional derivatives, and making use of high-level matrix operations to efficiently compute the derivatives. Furthermore MAD supports the use of MATLAB's `sparse` data-type to hold and propagate sparse derivatives allowing run-time sparsity exploitation – thus greatly enhancing performance in computing derivatives where sparsity is unknown or difficult to exploit via compression techniques. MAD allows computation of gradients and Jacobian matrices using the forward mode of AD. There is on-going work on implementing the reverse mode of AD, generating higher order derivatives and automatic sparsity detection. The MAD implementation is commercially licensed to TOMLAB [FE04]. The computational efficiency of calculating the derivatives with MAD was shown to be consistently better than ADMAT on several problems in Forth [For06]. Our MSAD implementation also inherits efficient derivative computations from MAD.

To test the benefits obtained by using source transformation together

with MAD's efficient data structures, we decided to gradually phase out the operator overloading from MAD. As proof of concept, this resulted in the tool MSAD [Kha04] that adopted a hybrid source transformation and operator overloading approach similar to ADiMat. This showed significant speedup for smaller test cases but asymptotically reached the performance of MAD as the problem size was increased.

1.4.2 Performance Issues

To make programming intuitive, MATLAB assigns a variable's class *implicitly* when it is first assigned, there is no explicit variable declaration. Attributes of variables such as class/type (integer, double, logical, complex, sparse, structure), shape (scalar, vector, matrix, array) and size (extents along each dimension) of variables can change as the program is run. Version 5.1 of MATLAB introduced object-oriented features that allowed users to define their own classes and associated operations. Together with overloading of functions and standard arithmetic operators, it became possible to seamlessly integrate these user classes with the predefined classes. Because there is no compiling phase before execution, the task of resolving class, shape of variables and hence the applicable functions, has to be deferred until run-time implying overheads in execution. Some key sources of overheads in the run-time performance of MATLAB programs have been isolated and detailed in [MP99]. We briefly re-emphasise these issues in the context of AD:

1. MATLAB executes operations using a *type check* and *dispatch* mechanism. The interpreter performs a run-time check on the class and shape attributes of the operands, then executes the appropriate function to compute the results. This is necessary because the variables in the original program are implicitly declared and untyped. The expression `C = A \ B` could therefore be a simple scalar division or a complex matrix linear solve depending on the class and shape of operands `A` and `B`. This forms a significant overhead, especially if the computational time of the operation is smaller than the check and dispatch time. AD implemented using operator overloading relies on this execution mech-

anism to carry out the differentiation and hence suffers most from these overheads.

2. *Arrays can grow dynamically* in number of dimensions, and in extents along any given dimension. To enable this feature, arrays undergoing resizing are internally assigned a new larger memory section and copied across completely. Consider the operations,

```
A = zeros(2,3);
A(2,2,1:4) = rand(1,4);
```

in which an initial rank 2 array has a third dimension added, increasing its size from `[2,3]` to `[2,3,4]`. The array `A` is assigned a new memory location that can hold `24` elements, and the previous `6` data elements are copied over into it. For larger arrays this involves a tremendous overhead as memory operations are several orders of magnitude slower than CPU operations. Active variables in forward mode AD generated code store n directional derivatives per element of an array. If the size of such an active array is increased to m by an assignment operation, the augmented operation involves allocating a new array of $m \times n$ elements and initialising these $m \times n$ elements.

3. MATLAB also enforces strict *array bounds checking*. An indexing operation on the right hand side of an expression with an index exceeding the current array dimensions or extents causes an error. As illustrated above, a similar operation on the left hand side grows the array to accommodate the elements outside the current extents. To accomplish this all indices used in the indexing operation need to be checked against the array bounds. To illustrate the point, and continuing the previous example, consider the following three semantically equivalent indexing operations on array `A`.

```
(1)                  (2)                  (3)
B = A(1,2:3,:);      B = A(1,2:3,1:4);    indx = 1:4;
                                          B = A(1,2:3,indx);
```

Operation `(1)` selects elements along the first row, second and third columns, and all in the third dimension. Note the use of the `':'` operation that implicitly selects all indices along that dimension in an indexing operation. Operation `(2)` uses a more explicit syntax `'start_indx:end_indx'` to select the indices in the third dimension. And, operation `(3)` uses a variable `'indx'` that holds all four index values, as an index for the third dimension. Although all three operations are semantically equivalent, the first is most efficient in terms of bounds checking because the indices used along the third dimension are implicit and known to be the array extents along that dimension, hence are not be checked at run-time. The second operation requires the `'start_indx'` and `'end_indx'` values to be checked if within bounds. The third operation requires all the indices held in the variable `'indx'` to be checked individually at run-time.

The complexity of index checking during AD is more than doubled, as the indexing operation is performed on the original array as well as the derivative array. The more acute problem is the indexing operation itself, since a derivative array has an extra dimension, retrieval of sub-arrays become computationally more complex. Indexing operations often form a performance bottleneck if the original function is coded badly.

4. To allow variables to change class, shape and size at run-time, MATLAB must allow dynamic allocation of memory for variables. Although the heap allocation strategy used in dynamic allocation makes bookkeeping (tracking and control of memory) of variables more efficient, frequent allocation and deallocation of variables of different sizes fragments the heap and memory allocation overheads increase thereafter. These run-time effects of memory management are very different compared to programs in typed languages that are compiled before execution for which memory for explicitly declared variables is allocated using frames on a stack, so eliminating any overheads. In addition to other sources of overheads listed earlier, we also believe that memory

effects due to dynamic allocation are significant.

1.4.3 Advantages using source transformation AD

"Compile-time resolution of overloading in languages such as APL and SETL has the potential for improving the run-time of programs."

– Bauer and Saal 1974

The primary aim of using source transformation for AD is to enable high level source code optimisations outside the scope of operator overloading. The MAD package for AD clearly demonstrates the efficiency and robustness obtained by employing efficient data-structures for derivatives and associated derivative combination operations through operator overloading. Using source transformation it is possible to extend this efficiency by exploiting inter-operation redundancies and use of available auxiliary information on shape, size and class of variables to optimise the complete augmented function. *Activity analysis* identifies the intermediate variables that are dependent on the independent inputs, the rest of the intermediates are constants with respect to the independents, and hence are assumed to have zero derivatives that need not be computed. Activity analysis carried out during the analysis and transformation phases therefore helps eliminate redundant derivative combination operations, or run-time checks to avoid such redundant operations carried out per operation.

As mentioned earlier, MATLAB programs are typically executed by an interpreter in lieu of the traditional compile and execute approach. Code optimisations mentioned in Section 1.3.1 that are relevant to AD, are available through an optimising compiler for languages such as C/C++ and Fortran, but are not available to augmented AD programs in MATLAB. Just-In-Time (JIT) compilation was introduced from MATLAB version 6.5 to reduce the overheads of program interpretation, but this has only limited applicability. JIT accelerates MATLAB code by converting conforming code into native machine code. As of release v7.12 (R2011a), *conforming code* includes loops and operations involving 2-, and 3-D arrays of native data types such as double, integer and logical only. Condition checks reducing only to scalar logical

types are accelerated. Since JIT compilation is applied online, aggressive compiler optimisations like common subexpression elimination (CSE) and constant propagation are dropped to avoid latencies. This fact was only experimentally verified, as `MathWorks` does not publicly document details of the optimisations applied.

Profiled runs of several test cases like the Brown's problem [Mat11a] and MINPACK `dgl2` [Len05] showed a significant portion of the time was spent on indexing operations occurring both on the left and right hand sides in an expression. To estimate the overheads arising from indexing operations, we emulated the CSE optimisation commonly performed by an optimising compiler. Figure 1.1 shows the original gradient function of the Brown's problem. Note the multiple occurrences of the vector indexing operations `x(i+1)` and `x(i)` on the right hand side, while array `x` remains unchanged. Looking for more commonality we can also spot the expressions `x(i+1).^2` and `x(i+1).^2` that occur more than once. To emulate CSE optimisation we construct another program given by Figure 1.2, that factors out the intermediate indexing and exponentiation operations and stores the values in variables `xi`, `xipl1`, `xipo2` and `xipl1po2`, and replace the uses by the appropriate copy. Table 1.6 lists the run times of the original function and the factored version of the function. On average we see a 20% decrease in run-time across problem sizes with the indexing operations factored out.

Table 1.6: CPU($\mathbf{g}$) (s) – CPU time (s) with & without CSE, and speedup CPU(non-cse)/CPU(cse) for gradient function of the Brown problem (2000 runs)

	CPU($\mathbf{g}$) (s) for problem size n							
Function	128	256	512	1024	2048	4096	8192	16384
`gbrown`	2.60	4.71	8.77	16.75	32.71	65.31	131.78	292.28
`gbrown (cse)`	2.00	3.77	7.28	14.24	28.18	56.12	111.70	241.38
Speed-up	1.30	1.25	1.20	1.18	1.16	1.16	1.18	1.21

```
function g = gbrown(x)
    n = length(x);
    g = zeros(n,1);

    i = 1:(n-1);

    g(i) = 2*(x(i+1).^2+1) .* x(i) .* ...
            ((x(i).^2) .^ (x(i+1).^2)) + ...
            2*x(i) .* ((x(i+1).^2) .^ (x(i).^2+1)) .* ...
            log(x(i+1).^2);

    g(i+1) = g(i+1) + ...
            2*x(i+1) .* ((x(i).^2) .^ (x(i+1).^2+1)) .* ...
            log(x(i).^2) + 2*(x(i).^2+1) .* x(i+1) .* ...
            ((x(i+1).^2) .^ (x(i).^2));
```

Figure 1.1: Brown function

```
function g = gbrown(x)
    n = length(x);
    g = zeros(n,1);

    i = 1:(n-1);

    xi = x(i);
    xipl1 = x(i+1);
    xipo2 = xi.^2;
    xipl1po2 = xipl1.^2;

    g(i) = 2*(xipl1po2+1) .* xi .* (xipo2 .^ xipl1po2) + ...
        2*xi .* (xipl1po2 .^ (xipo2+1)) .* log(xipl1po2);

    g(i+1) = g(i+1) + ...
        2*xipl1 .* (xipo2 .^ (xipl1po2+1)) .* log(xipo2) + ...
        2*(xipo2+1) .* xipl1 .* (xipl1po2 .^ xipo2);
```

Figure 1.2: Modified Brown function (emulating CSE optimisation)

Several other research efforts have been made to speed up MATLAB code, by translating programs to Fortran [RP99], through just-in-time compilation [AP02], partial evaluation [ELC03], and applying high-level source-to-source transformations [MP99]. These methods consider some of the MATLAB performance issues mentioned in Section 1.4.2, and suggest means to tackle them through source transformation. Chapter 1.4.1 discusses some of these approaches in more detail. With regards to AD, source transformation would therefore facilitate combining such compiler techniques with the existing AD techniques to build both robust and efficient AD tools for MATLAB and MATLAB-like environments.

The very first version of MSAD [Kha04] implemented MATLAB AD as part source transformation and inlining, and part operator overloading of MAD classes. Compared to MAD the results showed large improvements for small sized problems but the gains diminished with increasing problem size. This characteristic was attributed to the remaining layer of overloaded functions that performed derivative combinations, and the surplus temporaries in the augmented code. In the following version [KF06] we reported performance figures from MSAD implemeted as complete source transformation of input prgrams by inlining both `fmad` and `derivvec` class overloaded operations from MAD prior to execution. The results indicated significant improvements even with large problem sizes, and up to an order of magnitude improvement on small problem sizes. Because the generated code uses only intrinsic data-types and conformable program constructs, it also benefits from the underlying MATLAB JIT acceleration. The report also presented a case for applying compiler like optimizations to augmented MATLAB AD programs, with further savings of over 42% on test cases involving repeated array indexing operations and when applying common sub-expression elimination (CSE) only to indexing operations. This is consistent with the example above which shows a 20% improvement in run-time of just the function in Figure 1.2.

Other optimizations we believe are similarly relevant to performance of MATLAB AD and that can be applied in a source transformation framework are:

- Function inlining
- Constant propagation
- Branch optimizations
- Full- and partial- redundancy elimination
- Algebraic simplifications and reassociation
- Loop-optimizations (invariant code motion, strength reduction)

1.5 Goals and Requirements

The intent of MSAD here is to demonstrate the use of compiler techniques to apply AD to MATLAB programs. MSAD leverages on the `fmad` and `derivvec` classes, provided by the MAD [For06] package, to provide correct and efficient augmented operations implementing the forward mode of AD. Our previous efforts [KF06] demonstrate that complete source transformation by resolving, specialising and inlining overloaded MAD classes is feasible and profitable. However, this implementation of MSAD suffers from three drawbacks. The underlying *syntax directed translation* infrastructure [Kha04] of the software does not support control- and data-flow analysis. The lack of control and data information prevents: the processing of program constructs like loops and conditions; re-differentiability of programs to obtain second order derivatives; the possibility to apply the reverse mode of AD; and application of more complex compiler optimisations important in the context of MATLAB AD in Section 1.4. Secondly, the intermediate representation (IR) is based on a simple abstract syntax tree (AST). Although the representation is simplified to a three address form (two, or one operands and one result) the operands and the results may be indexed MATLAB array values. The complex intermediate representation hinders our ability to process nested data-structures like structures and cell arrays. And finally, the core `fmad` and `derivvec` class routines are hard-wired into the software which makes enhancement to the AD operations and maintenance difficult.

Goals

1. Our primary goal is therefore to provide a new infrastructure for MSAD with better coverage of MATLAB constructs, which is also simple yet extendible allowing modification or addition of new AD operations and algorithms.

2. The second and equally important goal is to provide a generic and extendible infrastructure that supports high level optimisations and transformations of MATLAB programs based on state-of-the-art compiler techniques. And to be able to apply a similar optimisation approach to other potential applications which may rely on operator or function overloading such as interval analysis, propagating arbitrary taylor coefficients, etc.

3. Third more broad goal is a design that favours re-targetability of the output to possibly another high-level language like C, or even compile to assembly in the future.

Whilst aiming to support more complex analysis, optimisations, and better AD algorithms, we wish to set some basic requirements of the AD tool.

Requirements

1. Readability of augmented code when targeting a high-level language. This is important is the user wishes to inspect, hand-tweak the code or manually fit the derivatives in a larger framework of software.

2. Correctness of augmented AD output.

3. Correctness of specialised output program, non-AD, for all inputs that fit the specified input constraints. This is the premise to correctness of the augmented AD code. We assume the input programs have been tested for correctness.

4. Reasonably small compile-time.

5. Self-contained and relative independence from third-party tools or libraries.

Chapter 2

Background

A compiler is commonly perceived to be a program that takes in an input source program coded in a high level language, and converts it to a low level assembly language, native to the target processor it is compiling for. A more general view of a compiler is a program that reads a program written in one *source* language, and translates it into an equivalent program in another *target* language [ASU86]. A more pertinent definition of modern compilers by Muchnick [Muc97] is as follows:

Definition 2.1 (Compiler) *A compiler consists of a series of phases that sequentially analyze given forms of a program and synthesize new ones, begining with the sequence of characters constituting a source program to be compiled and producing ultimately, in most cases, a relocatable object module that can be linked with others and loaded into a machine's memory to be executed.*

A typical compiler processes an input program in several phases, as shows in Figure 2.1, each adding to or transforming the representation of the input program. To carry out the compilation process in a systematic manner, the compiler uses layers of abstraction to help delineate between data, information and semantics of the program. Many auxiliary functions need to be provided to assist the above process and which help store and retrieve any intermediate or associative information. A compiler may choose to carry out each of the processing phases over an independent run or pass of the input

program. Note that there are compilers that operate in a single pass of the input, but most modern compilers tend to use several passes to complete the translation process. The compilation process is often split into two stages, the compilation phases that ultimately result in a sequence of *mnemonics* or an assembly language representation of the program, followed by an assembly stage that translates the mnemonics to object code. All the necessary objects are later linked together by a linker into an executable.

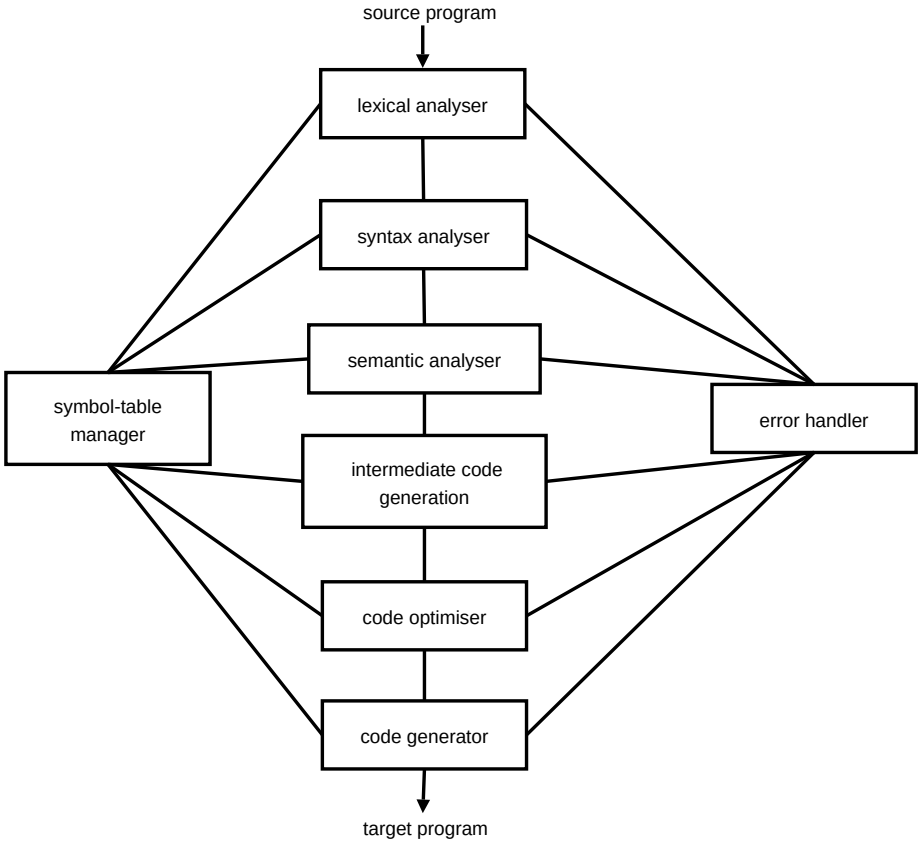

Figure 2.1: High-level structure of a compiler

In addition to translating a high-level language to assembly, it is often implicit that the compiler translates it to an optimised assembly program. Ideally the assembly program would be as good as written by hand, but in practise it may not be as optimal. There are two main reasons, the compiler only solves the problem of translating the high-level program to an effi-

cient binary executable. It does not solve the software engineering problem of choosing the correct data-structures or algorithms. Even if it is assumed that the program in the high-level language is optimally designed, the several problems that the compiler needs to solve are NP-complete [ASU86, Ch.9] (register allocation, instruction scheduling, code and data layout) or otherwise very computationally expensive. Most compilers use approximations or heuristic algorithms to solve these problems. Aho, Sethi and Ullman [ASU86, Ch.10] define an optimising compiler as follows:

Definition 2.2 (Optimising Compiler) *Compilers that apply code-improving transformations are called optimising compilers.*

There are two main models of optimising compilers, a *low-level model* that uses a low-level *intermediate representation* (IR) to perform all optimisations, or a *mixed model* that uses a medium-level IR (MIR) to perform high-level optimisations, and a low-level IR to perform machine dependent optimisations. The reason to use an intermediate representation is covered in Section 2.2.1. For the purpose of AD we translate the high-level source language to a medium-level IR, and apply optimisations and AD transformation to the MIR.

2.1 Basic Phases in Compilation

2.1.1 Lexical Analysis

Lexical analysis [ASU86, Ch.3] involves reading the raw source character stream from left to right and grouping the characters into character strings or *lexemes* that have an associated meaning. All lexemes with the same meaning fall under a single class and are assigned a *token*. A token may be an integer or be simply represented by a string of capital letters. Consider variables `a`, `b` and `pi`, used in a program. During lexical analysis the variables names `a`, `b` and `pi` are stored as pairs of lexemes and associated tokens. In this example the lexemes will be `a`, `b`, `pi`, and the associated tokens may

be `IDENTIFIER`, `IDENTIFIER`, `CONSTANT` respectively. This process is termed *scanning*.

Lexical analysers are based on pattern matching algorithms that allow efficient scanning of the input data. Generally lexemes in the input stream are associated with patterns specified using *regular expressions* [Aho90] over the language alphabet. The regular expression for any input of the token type `IDENTIFIER`, in the earlier example, may be defined as `([a-zA-Z]([a-zA-Z0-9])*)`. This implies that the first character of the variable name may be a case insensitive character, followed by any number of such characters or digits. A group of connected definitions of such regular expressions forms a regular definition. If we define the regular expressions for characters using `L -> [a-zA-Z]`, and digits using `D -> [0-9]`, then the `IDENTIFIER` token is given by the regular expression `L(L|D)*`. These expressions together form a regular definition.

Conventionally these are converted to an NFA, *non-deterministic finite automata* and further reduced to a DFA, *deterministic finite automata* before being coded-up into a programming language to produce a scanner. A *finite automaton* is a mechanism by which regular expressions can be matched with the presented input. The NFA is conceptually closer to the regular expressions and hence the regular definitions are first converted into an equivalent NFA. The DFA is a more practical representation for implementation and hence, using well-known algorithms [ASU86], this is converted into an equivalent DFA.

The data file to be parsed is generally present on a physical disk that is several orders slower than the CPU that processes the data. To make the scanning efficient methods like double buffering and prefetching are often adopted [ASU86]. Several tools like *Lex* [LS] automate this entire process and allow scanners to be generated directly from the specifications of regular definitions or regular grammar.

2.1.2 Syntax Analysis

Syntax analysis involves grouping tokens hierarchically into nested collections with collective meaning. In our case these nested collections form constructs supported by a programming language, and they tend to have an inherently recursive structure. This makes it essential to have a *grammar* that is sufficiently general, to be able to define their syntax. *Context-free grammars* are such essential formalisms used for describing the structure of programs. In the context of a parser, we use a token and a lexeme interchangeably, since the parser, by itself, does not differentiate between two inputs with the same token type.

A grammar is specified in terms of *production rules*, that have a *non-terminal* on the left-hand side and a combination of *terminals* and non-terminals on the right. A terminal is essentially a token or a lexeme supplied by the lexer, and a non-terminal is a combination of many such tokens, or non-terminals. Note the inherent recursion in the definition of a non-terminal. Figure 2.2 gives a sample production rule for an arithmetic `expression`. This standard format of presenting the context-free grammar is called the *Backus-Naur* Form or BNF. In these productions `IDENTIFIER`, `+, -` are terminals and `expression`, `prefix_expr` are non-terminals. We note the use of recursion in this definition by including the left non-terminal, on the right side.

```
expression  -> expression '-' expression
             | expression '+' expression
             | prefix_expr

prefix_expr -> '-' prefix_expr
             | '+' prefix_expr
             | IDENTIFIER
```

Figure 2.2: Expression production

The grammar in Figure 2.2 can match expressions of the form in (2.1). The variables in these examples, `a`, `b`, `c`, `d`, `var`, `temp`, are assumed to be

assigned the token `IDENTIFIER` by the scanner.

```
a + b - c
a - -b + temp                    (2.1)
a - -var + d + +temp
```

BNF is suitable for expressing nesting and recursion, but is less convenient for expressing repetition and optionality [PLW+04]. An *Extended BNF* allows such constructs through the use of three postfix operators, `+`, `*`, `?`. A sample production rule that matches arguments to a function call is given in the Figure 2.3. This rule matches an opening parenthesis followed by any number of identifiers separated by commas followed by a closing parenthesis. It also allows for the case where no identifiers are present within the enclosing parentheses.

```
def_arg_list -> '('
                  ( identifier ( COMMA identifier )* )?
                ')'
```

Figure 2.3: Function arguments production

There are two main classes of parsing techniques, *top-down* parsing and *bottom-up* parsing. Top-down parsers begin with the start symbol of the grammar and attempt to find a leftmost derivation for an input string of tokens. The practical implementations of this type of parsers use a *recursive-descent* method to follow the productions in a grammar. One such method is the **LL(*k*)** or the left-to-right parsing using the leftmost derivation with k symbols lookahead. To avoid the use of backtracking, the grammar needs to be converted to another equivalent form by eliminating any left recursion and applying left factoring to avoid a large lookahead. This will be looked at in detail in Chapter 3. For now we assume that the grammar in Figure 2.2 is converted to an equivalent form in Figure 2.4, left factored and free of left recursion.

The top down parsing of the second expression of (2.1) with single lookahead ($k = 1$) according to the grammar in Figure 2.4, is shown in Figure 2.5,

```
expression  -> prefix_expr          (1)
               ( '+' prefix_expr    (2)
               | '-' prefix_expr    (3)
               )*

prefix_expr -> '-' prefix_expr      (4)
             | '+' prefix_expr      (5)
             | IDENTIFIER           (6)
```

Figure 2.4: Expression production

as the input is read left to right. LA gives the lookahead character and input position is the position of a token in the input stream.

Input pos.	Input lexeme	LA	Production suggested	Matched expression
0		a	(1)	`expression`
1	a	$-$	(6,3)	`prefix_expr`
2	$-$	$-$	(4)	`ID '-' prefix_expr`
3	$-$	b	(6)	`ID '-' '-' prefix_expr`
4	b	$+$	(2)	`ID '-' '-' ID`
5	$+$	*temp*	(6)	`ID '-' '-' ID + prefix_expr`
6	*temp*			`ID '-' '-' ID '+' ID`

Figure 2.5: Parsing steps for (2.1)

Bottom up parsers on the contrary attempt to reduce substrings of tokens to the starting symbol of the grammar, effectively working their way up from the leaves of the parse tree towards the root. At each reduction step a particular substring matching the right side of a production is replaced by the symbol on the left of that production. Parser implementations of this type include $\mathbf{LR}(\boldsymbol{k})$ left-to-right parsing using rightmost derivation, and the $\mathbf{LALR}(\boldsymbol{k})$ which is **LR** with k symbols lookahead. These methods used a table based approach to perform the reductions efficiently. We will not look into the details of this methods since we use top-down parsing in our implementation.

2.1.3 Semantic Analysis

The result of lexical and syntax analysis is generally a *parse tree* or an *abstract syntax tree*. Consider the grammar to match a small subset of arithmetic expressions defined by the production rules in Figure 2.6.

```
statement  -> identifier '=' expression
expression -> expression '+' expression
            | prod_expr
prod_expr  -> prod_expr '*' prod_expr
            | '(' expression ')'
            | identifier
            | constant
identifier -> IDENTIFIER
  constant -> REAL
            | INTEGER
```

Figure 2.6: Simple arithmetic expression grammar

Figure 2.7 shows the parse tree generated for this grammar on the simple input (2.2)

```
c = A * x + b                                    (2.2)
```

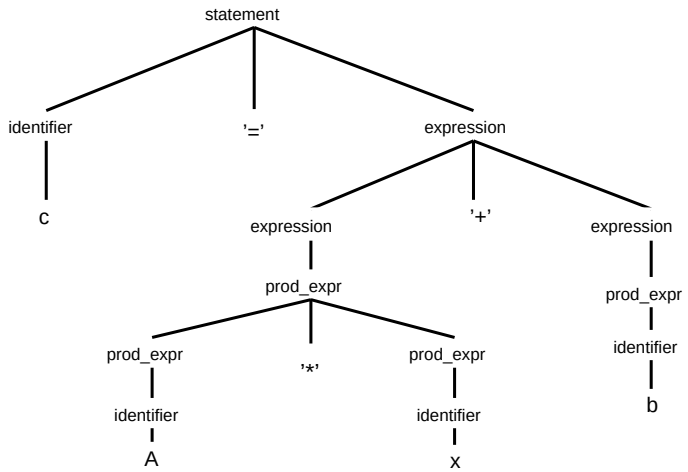

Figure 2.7: Parse Tree for (2.2)

The AST or the abstract syntax tree in Figure 2.8, gives a more compact representation neglecting all the production steps. At this stage in compila-

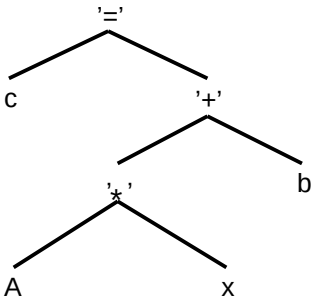

Figure 2.8: Abstract Syntax Tree for (2.2)

tion, the nodes in this tree typically hold minimal information such as the lexeme and token for each terminal.

During the semantic analysis phase, knowledge of the underlying operation is gathered and stored as annotations in the syntax tree. Consider that statements (2.3 - 2.6) are parsed according to the grammar in Figure 2.6.

```
A = 30.6          (2.3)
x = 5             (2.4)
b = 8             (2.5)
c = A * x + b     (2.6)
```

We assume here that the lexical analyser assigns the tokens `REAL`, `INTEGER` and `INTEGER` to the constants `30.6`, `5` and `8` respectively. We also assume that the parsing phase matches the syntax of the input expressions to the grammar and constructs a syntax tree for each of the assignment statements. The first assignment in (2.3) then says that a real number is assigned to the variable `A`, the semantic analysis phase then sets the *type* attribute for the variable `A` to `REAL`. Similarly variables `x` and `b` have their type attributes set to `INTEGER`. We identify (2.6) to be the same as (2.2), hence the syntax tree generated by the parser will be as in Figure 2.8. The semantic analysis phase then proceeds to do a similar type inference for the variable `c` using this tree. Since `A` is a `REAL`, and `x` an `INTEGER`, the result of the operation will be a `REAL`, hence the *type* attribute for the operation '`*`', is set accordingly. Using

a similar reasoning, we find that the *type* for the operation '=', is `REAL`. This is called attribute *synthesis*. Further, since `c` is equal to the entire expression on the right, `c` inherits its *type* attribute `REAL` from '='.

Annotating the syntax tree [GBJL86] may be carried out either by the use of *attribute* grammars or by *manually* traversing the tree. Attribute grammars allow code for context-handling to be generated from a high-level specification similar to that used for generating the parser, though not many implementations support attribute grammars completely. It is also possible to incorporate context handling inside the context free grammar that is used for parsing, which is adopted by several current implementations of parsers [DS95, PQ95]. This requires two extensions to the context free grammar, one for storing the attribute data and the second to allow the evaluation of the attribute values. With each terminal or non-terminal we associate zero or more *formal attributes*, allowing distinct instances of these to posses the same attributes but perhaps different values. Also, associated with each production rule in the grammar, is a set of attribute evaluation rules that compute the attributes of the current symbol in terms of the attributes of one or several other symbols. Moreover the attributes associated with any symbol may be synthesised or inherited attributes. A system that makes use of these constructs to compute the attribute values is called an *attribute evaluator*.

Equivalently a more restricted class of ordered attribute grammars, *L-attributed* or *S-attributed*, may be used to make the attribute evaluation more tractable. The type of parsing method used favours a particular grammar. Top-down parsing lends itself to the the use of L-attributed grammars with inherited attributes, and bottom-up to S-attributed with synthesised attributes, but essentially both these grammar types are equivalent. Although we do not use *attribute grammars*, we use advanced iterative algorithms like *Sparse conditional constant propagation* (SCCP) [WZ91] that are control-flow aware to propagate attributes in our current implementation. Nonetheless the basics of *attribute synthesis* are a precursor to understanding these algorithms.

Not all semantic analysis relates to variables. Information is very of-

ten represented hierarchically to abstract out properties such as change of control-flow. Contiguous statements that always execute sequentially may be grouped together into a *basic-block* [ASU86, Ch.9] and useful properties attached to this basic-block. Special analysis based on *data-flow equations* [ASU86, Ch.10] is required to propagate these properties. These concepts will be covered in detail in the following sections.

2.1.4 Code Generation

In this final phase the compiler converts the, possibly optimised, low-level intermediate representation to efficient target code. The problem of generating optimal code is mathematically undecidable [ASU86, Ch.9]. However, the compiler uses heuristic techniques to map the intermediate level constructs to concrete machine level constructs like instructions implementing correct operations and appropriate data-widths (*instruction selection*), registers satisfying any data-dependencies and spilling contents to memory if required (*register allocation*), and correct ordering of instructions (*instruction scheduling*). Conventionally all compiler techniques target a low level assembler mnemonics and hence consider the code generation issues mentioned earlier. In this particular application we are concerned with turning the intermediate code back to MATLAB. We will therefore not delve into conventional code generation techniques in further detail.

2.2 Supporting infrastructure

2.2.1 Intermediate Representation

The first three phases listed in Section 2.1 form the front-end of a compiler, and are collectively responsible for analysing the input program for its syntax, semantics and gathering any useful information for the remainder of the compilation phases. The phases at the core of the compiler tend to focus on the logic of the input program rather than the syntax, and hence prefer to use a native format or language that is more suitable than either the source

or the target languages. Intermediate code generation usually increases the size of the AST, but it reduces the conceptual complexity and this gives a two-fold advantage. It makes re-targeting possible, i.e. the same compiler can be made, at a later date, to generate an output language that is different from the source one, without much rework. Additionally, the compiler can apply several optimisation techniques to this simple yet equivalent form of the input program. There are several kinds of intermediate representations that are used in compilers, a few of these include linked-lists, an intermediate code tree and three-address statements.

Intermediate code trees are similar to ASTs, with the difference that the code trees are independent of the input language syntax. Since ASTs are generated by the parser, the structure is still tied to the syntax of the input language. Intermediate code trees are built from ASTs by restructuring, stripping them of any source syntax and creating a clean tree that is easier to operate on. The three-address statement employs essentially the same strategy but by breaking statements down into binary operations. Since the arithmetic operations are unary or binary, they can straightforwardly be converted into three-address plus operation *Quadruples* form, with appropriate temporaries holding the intermediate results. Consider the MATLAB statement in Figure 2.9 which computes the result of an expression. Figure 2.10 gives the corresponding AST.

```
r23 = ((x - mustar)^2 + y^2)^1.5;
```

Figure 2.9: Statement computing the result of an expression

Each binary operation is then separated into individual expressions and the temporary result used in the following expression. Thus, Figure 2.11 forms the Quadruples representation of the statement in Figure 2.9.

Control flow statements, like loops and conditionals can also be converted to this form using abstract commands like `jump` or `goto` with labelled addresses for targets. Consider the MATLAB code fragment in Figure 2.12 which uses a `for` loop to execute statement `3` ten times. Figure 2.13 shows

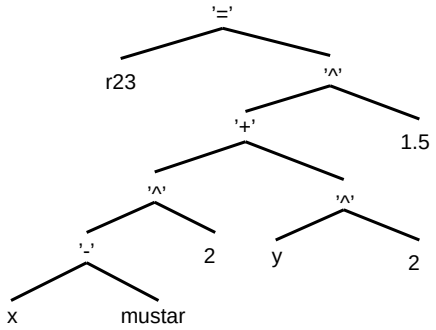

Figure 2.10: AST for statement in Figure 2.9

```
1       tmp_1_ = x - mustar
2       tmp_2_ = tmp_1_ ^ 2
3       tmp_3_ = y ^ 2
4       tmp_4_ = tmp_2_ + tmp_3_
5       tmp_5_ = tmp_4_ ^ 1.5
6       r23 = tmp_5_
```

Figure 2.11: Quadruples intermediate form for statement in Figure 2.9

an equivalent intermediate representation for this code. Note that the `for` loop has been reduced to explicit test operation in statement `(3)`, and two jump operations using a `goto` in statements `(3)` and `(9)`. The remaining expressions have been simplified to the Quadruple form like in the previous example. The loop index is initialized in statement `(2)` before the loop body, and incremented by the stride, in this case one, in statement `(8)` within the

```
1       y = 1.618;
2       for i = 1:10
3           y = y + (y / (pi * i));
4       end
5       display(y);
6       display(i);
```

Figure 2.12: Code fragment with change in control-flow

loop. This explicit control-flow structure makes it easier for the compiler to analyse and optimise each statement.

```
1           y = 1.618
2           tmp_1_ = 1
3       L1: if tmp_1_ > 10 goto L2
4           i = tmp_1_
5           tmp_2_ = pi * tmp_1_
6           tmp_3_ = y / tmp_2_
7           y = y + tmp_3_
8           tmp_1_ = tmp_1_ + 1
9           goto L1
10      L2: display(y)
11          display(i)
```

Figure 2.13: Intermediate form for code fragment in Figure 2.12

2.2.2 Symbol Table management

A *symbol table* is a data structure containing a record for each identifier, with fields for the attributes of the identifier, together with its scope and binding information [Muc97, Ch.3]. This record is accessible by performing a lookup using the name of the desired identifier. Entries are added to the table during scanning or parsing if a new symbol, not present in the table, is encountered. Very often all the information about a symbol is not available at once, hence the symbol record in the table is updated as and when information is made available. Symbol information is accessed frequently during the compilation phases and to ensure a reasonable efficiency in this operation the data-structures used to build the table require careful consideration. Moreover the number of symbols that need to be stored in the table is not known in advance, this rules out the use of statically allocating space for symbols in the table and hence dynamic allocation strategies are used instead. Data-structures [AHU74, Knu97] like *hash-tables*, *lists*, *sets*, and *bit-vectors* or *lattices* are often used. We will look at lattices in detail in Section 2.3.1.

Symbols encountered during parsing may be of several different *types*

(scalars, arrays, functions), and have different formal attributes associated with them. For example, an identifier representing a scalar variable may have attributes such as *data type*, or *is constant*, but an identifier representing a function might have attributes such as *number of arguments*, *number of results*, or *local symbol table*. Since the physical record structure, storing these attributes, then cannot be of a fixed type, the individual records in practice are allocated externally and in place of the records, their references are stored in the symbol table.

Scope handling

A variable is accessible within a programming language construct only if it is present in the *scope* of that construct, and most languages support nesting of constructs and hence the scopes. This is handled by maintaining a symbol table for each separate scope or marking the scopes with labels and storing these along with the attributes to an identifier in its record.

2.3 Semantic Analysis and Optimisation

In this section we introduce the necessary compiler concepts to appreciate the MSAD implementation, and differentiate between approaches taken by other tools to perform AD, or to optimise MATLAB code.

2.3.1 Attribute Inference

Section 2.1.3 introduced the concept of attribute synthesis. Every symbol in MATLAB possesses intrinsic properties which are formally called *attributes*. We identify the following minimal set of attributes necessary for specialising and optimising MATLAB programs. The complete set of attributes necessary may vary depending on the problem to be solved. We are concerned with generically optimising MATLAB code with AD as one application.

Type

Because MATLAB was originally designed to be an interpreted environment with the intent to simplify the modelling of complicated mathematical formulations and algorithms without having to deal with underlying storage representation, the MATLAB language was chosen to be *dynamically typed*. Unlike *statically typed* languages in which the type of every expression can be determined at compile time (using programatic declarations and static program analysis), dynamically typed languages postpone the resolution of the exact type of an expression until run-time.

Every symbol in MATLAB can potentially be a function, or an array of homogenously or heterogenously typed objects. An array of homogenously typed objects could be an array of one of the following types *integer* (8-, 16-, 32- or 64-bit, signed or unsigned values), *real* (single or double precision floating point values), *logical*, *character*, *function handle* or *user-defined classes* [Mat11b]. Each element of a heterogenously typed object array can be any of the previously mentioned types. Based on how each element in an heterogenously typed array is accessed there are two types in MATLAB, *cell* that are indexed by numeric values, and *structure* that are indexed by field names.

Definition 2.3 (Type inference) *Type inference is the problem of determining the type of a language construct from the way it is used.*

Determining the exact type for all symbols without programatically declaring every symbol in a program is mathematically undecidable [Wel93]. However, the problem here is to be able to determine the possible types of an expressions in a dynamically typed language like MATLAB at compile time. There are two aspects to solving this problem. First we need a mechanism to represent the possible types of any given symbol. During the process of inference we may determine that a symbol has one fixed type, or a set of potential types. The type representation should be able to cope with this. Secondly, we require a method to infer the type of a symbol from the context in which it is used. This includes infering the type of a result from its

operands or the syntax, or both.

Ideally, the type representation should be able to provide the following operations on the type of a symbol:

1. Query whether the symbol has a fixed type.
2. If not fixed, query the smallest *type-set* that all the potential types, of the symbol, belong to.
3. Query if potential types includes another type.
4. Narrow the type based on another type or set of types.
5. Widen the type based on another type or set of types.

A concept often used for various static analyses, including type inference, in a compiler is a *lattice* [RMG+00]. The following section introduces a lattice and its uses in relevant analyses and optimisations.

Lattice

DEFINITION 2.4 (LATTICE) *A lattice* $(\mathbf{L}, \vee, \wedge)$ *is a nonempty set* $\mathbf{L}$ *closed under two binary operations* $\vee$ *(join) and* $\wedge$ *(meet) such that the following laws are satisfied for all* $\mathtt{a}, \mathtt{b}, \mathtt{c} \in \mathbf{L}$*:*

associative laws:	$\mathtt{a} \vee (\mathtt{b} \vee \mathtt{c}) \equiv (\mathtt{a} \vee \mathtt{b}) \vee \mathtt{c}$	$\mathtt{a} \wedge (\mathtt{b} \wedge \mathtt{c}) \equiv (\mathtt{a} \wedge \mathtt{b}) \wedge \mathtt{c}$
commutative laws:	$\mathtt{a} \vee \mathtt{b} \equiv \mathtt{b} \vee \mathtt{a}$	$\mathtt{a} \wedge \mathtt{b} \equiv \mathtt{b} \wedge \mathtt{a}$
absorption laws:	$\mathtt{a} \vee (\mathtt{a} \wedge \mathtt{b}) \equiv \mathtt{a}$	$\mathtt{a} \wedge (\mathtt{a} \vee \mathtt{b}) \equiv \mathtt{a}$

DEFINITION 2.5 (SUBLATTICE) $\mathbf{L}_1$ *is a sublattice of lattice* $\mathbf{L}$ *if* $\mathbf{L}_1 \subseteq \mathbf{L}$ *and* $\mathbf{L}_1$ *is a lattice using the same operations as those in* $\mathbf{L}$*.*

In addition to satisfying the above laws, a lattice also possesses useful properties that allow us to perform all the operations identified earlier for type inference. Some of the useful properties include:

1. Every lattice is a *partially ordered* set. This implies an *ordering relation* $\leq$ can be defined on the elements of the lattice set such that $\mathtt{a} \leq \mathtt{b}$ also means $\mathtt{a} \vee \mathtt{b} \equiv \mathtt{b}$ or equivalently $\mathtt{a} \wedge \mathtt{b} \equiv \mathtt{a}$. Or $\mathtt{a} < \mathtt{b}$ in the case of strict

ordering where $\mathtt{a} \neq \mathtt{b}$. In case of type lattices, this property allows testing if a type-element belongs to or a type-set is a subset of another type-set.

2. If $\mathbf{L}$ is a lattice and $\mathtt{a}, \mathtt{b} \in \mathbf{L}$, then $\mathtt{a} \wedge \mathtt{b}$ and $\mathtt{a} \vee \mathtt{b}$ are unique. This property allows narrowing and widening a type-set.

3. Every pair of elements in the lattice also have a *least upper bound*, $\mathtt{lub}(\mathtt{a}, \mathtt{b})$ and a *greatest lower bound*, $\mathtt{glb}(\mathtt{a}, \mathtt{b})$ which follows from the ordering relation. A lattice has two unique elements denoted by $\top$, a top, and $\bot$, a bottom element. The top is defined as $\forall x \in \mathbf{L}, x \leq \top$ and the bottom as $\forall x \in \mathbf{L}, \bot \leq x$. These elements usually form the boundary conditions, starting point of a type inference or error condition.

4. According to Definition 2.5 a subset of all the potential types is also a lattice by itself. This property is used to identify the smallest type-set that covers all the potential types, analogous to an least upper bound (LUB). Because the type-set is also a set, it is trivial to identify if it represents a unique type.

Consider the set $\mathbf{S} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. If we construct the power set $\mathcal{P}(\mathbf{S})$, a set containing every subset of $\mathbf{S}$, we get:

$$\mathcal{P}(\mathbf{S}) = \{\{\}, \{\mathtt{a}\}, \{\mathtt{b}\}, \{\mathtt{c}\}, \{\mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{c}\}, \{\mathtt{b}, \mathtt{c}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}\} \tag{2.7}$$

The set $\mathcal{P}(\mathbf{S})$ in equation 2.7 forms a lattice, shown in Figure 2.14, if we impose the ordering relation subset, $\subset$, and the meet and join operations to be set intersection, $\cap$ and set union, $\cup$ respectively. The top, $\top$ in this case is $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and the bottom, $\bot$ is $\{\}$. The $lub(\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{b}, \mathtt{c}\})$ for example is $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, the $glb(\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{b}, \mathtt{c}\})$ is $\{\mathtt{b}\}$.

Within a compiler a lattice is often implemented efficiently using *bit-vectors*. A bit-vector is simply a string of binary bits. The lattice in equation 2.7 can be represented using a three bit vector $\langle 000 \rangle$. The element $\mathtt{a}$ represented by $\langle 001 \rangle$, $\mathtt{b}$ by $\langle 010 \rangle$ and $\mathtt{c}$ by $\langle 100 \rangle$. The set operations union

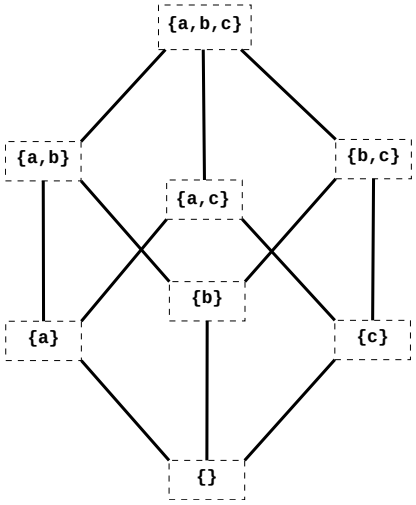

Figure 2.14: Hasse diagram for lattice in (2.7)

and intersection from the above example are replaced by bitwise-operations, `BIT-AND` for lattice meet and `BIT-OR` for join. Once the representation is established the remainder of the problem is propagating the lattice values.

Propagation of lattice values is commonly carried out using iterative techniques, iterating over statements in a program, until all the lattice values in the program are said to have reached a *fixed point*. To reach a fixed point the function computing the result lattice is required to be *monotone* [Muc97].

Definition 2.6 (Fixed point) *A fixed point of a function* $f : \mathbf{L} \to \mathbf{L}$*, is an element* $\mathtt{z} \in \mathbf{L} \mid f(\mathtt{z}) = \mathtt{z}$.

Definition 2.7 (Monotone function) *A function mapping a lattice to itself, given by* $f : \mathbf{L} \to \mathbf{L}$*, is monotone if* $\forall \mathtt{x}, \mathtt{y} \in \mathbf{L} \mid \mathtt{x} \leq \mathtt{y} \Rightarrow f(\mathtt{x}) \leq f(\mathtt{y})$.

We will look at iterative techniques and appropriate monotone inference functions in the later sections 2.3.3 in this chapter, and more implementation details in chapter 3. The type is the most significant of the other attributes in that the type determines other attributes that may be of interest. For example if a MATLAB variable is an array it has a certain *shape*, *rank* and *value*. If the array variable is of type double or logical it could have a *sparse* representation [Mat11b]. Independently, if the array variable is any of the

numeric types it could be complex, i.e. have *real* and *imaginary* parts. In case of heterogenous containers like structure and cell arrays, the value itself could have a recursive representation with arbitrary nesting of structure, cell, and numeric array components. A structure array, for example, has all the properties of an array listed above, along with a table of *field* names, which are common to all elements of the structure array. Additionally we would require an array of symbols, for each field, each element of which has all the properties listed above. We term this generic representation of types and attributes a *value representation*.

Shape

The *shape* of an array is a *tuple* of the extents of the array along each of its dimensions. Each element of the tuple also forms the upper bound on the index of that dimension. The lower bound on index in any dimension in MATLAB is implicitly one. An array `x` created after executing the statement `x = zeros(5,4,3,2)` will have the shape $\langle 5, 4, 3, 2\rangle$. Note MATLAB arrays are stored column major so the first element in the shape tuple represents the number of rows, second the columns. Almost all operations in MATLAB ignore trailing singleton dimensions, which implies the shape of the array `x` can also be equivalently represented as $\langle 5, 4, 3, 2, 1\rangle$, $\langle 5, 4, 3, 2, 1, 1\rangle$ etc. The *canonical shape* tuple is however $\langle 5, 4, 3, 2\rangle$. For simplicity we use shape synonymously with the canonical shape from here on. The minimum number of dimensions that an array must have is two. MATLAB also has a concept of an empty matrix `[]`, which has the shape $\langle 0, 0\rangle$.

```
1      x = rand(4,2);       x = pi;              x = A;
2      y = ones(2,3);       y = ones(2,3);       y = ones(2,3);
3      z = x * y;           z = x * y;           z = x * y;

           (1)                  (2)                  (3)
```

Figure 2.15: Shape inference example

In Figure 2.15 example `(1)` the array variables `x` and `y` have shapes $\langle 4, 2\rangle$

and $\langle 2, 3 \rangle$ respectively. The result `z` can be statically inferred to have the shape $\langle 4, 3 \rangle$. In example `(2)` however, variable `x` is a scalar and the result `z` has the same shape $\langle 2, 3 \rangle$, as variable `y`. In example `(3)` the shape of `x` cannot be statically determined, unless `A` is a known value. Here the shape of `z` can at best be determined to be $\langle ?, 3 \rangle$ under the assumption that `A` has a conformable shape with `y` for the multiply operation to apply correctly, and where '?' represents an undetermined value.

The shape attribute can also be represented by a lattice, however Joisha and Banerjee [JB06] point out the limitation of using shape lattices for shape inference is that the operations on shape tuples are not always monotonic. Alternately these authors point out that symbolic shape inference can be carried out pre-compile time using a *shape algebra*, which computes the equations governing the shape of the result of a MATLAB operation. These equations then compute the shape tuple during compile time. We programatically compute the shape tuple of the result, which in effect is similar to the symbolic equations.

Rank

The *rank* of an array variable is the number of dimensions of the array (number of elements in the shape tuple). The MATLAB equivalent of the rank of an array is the built-in function `ndims` which also returns the number of dimensions of its argument. It is simple to compute the rank of an array once the shape of result is computed through shape inference. However because we compute the shape numerically, any missing information of the shape translates to completely missing information of the rank. We therefore perform a separate rank inference which is based on both the result of the shape inference and a rank lattice. Figure 2.16 shows the rank lattice used in MSAD. Note this is a trivial lattice with all elements in a linear tree.

Sparsity

In MATLAB, array variables of type *double* or *logical* which are strictly two dimensional, can use a special *sparse* representation. If the values stored in

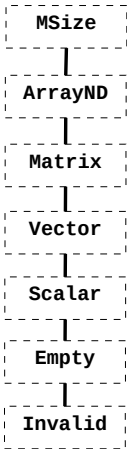

Figure 2.16: MSAD MATLAB array rank lattice

the array are known to be mostly zero-valued, the storage requirements using the sparse representation can be reduced from $m \times n$ to $O(\texttt{nnz}(\mathbf{S}))$, where the shape of the sparse matrix $\mathbf{S}$ is $\langle m, n \rangle$ and `nnz` is the number of non-zero elements in the array [GMS92]. For large sparse matrices this implies the size of the sparse representation is a small fraction of the default full representation.

To make handling of sparse variables transparent to the user, MATLAB automatically propagates the sparseness or *storage class* of operands in an operation to its results based on simple rules. Propagating sparseness implies at least one of the operands should be sparse. The exceptions to this rule are constructor-like functions `sparse`, `spones`, `speye`, `sprand`, etc. which are meant to create sparse results. The sparsity propagation rules [GMS92] are summarised below (S represents an operand with sparse storage, and F represents an operand with full storage):

1. Functions from matrices to scalar or fix-size vectors always generate full results, e.g. `size`, `nnz`, etc.

2. Functions from scalars or fixed-size vectors to matrices like `ones`, `eye` generally return full results with the exceptions listed earlier `spones`, `speye`, etc.

3. Unary functions return the same storage as the argument i.e. $f(S) \rightarrow S$ and $f(F) \rightarrow F$ with the exceptions `full` and `sparse`

4. In case of binary operations where both arguments are full or both sparse, the result is full or sparse respectively. If the storage class of the two differ the following rules apply:

$$S + F \rightarrow F \quad \text{and} \quad S * F \rightarrow F$$
$$S .* F \rightarrow S \quad \text{and} \quad S \,\&\, F \rightarrow S$$

5. Indexing operations preserve the storage class of the array variable:

$$T = S(i,j) \quad \Rightarrow \quad T \rightarrow S \quad \text{if either } i \text{ or } j \text{ is a vector}$$
$$T(i,j) = S \quad \Rightarrow \quad T \quad \text{retains sparse or full storage}$$

6. For concatenation operations, `vert-`, `horz-cat`, if any of the operands is sparse the result is sparse, otherwise the result is full.

As a symbol attribute, sparseness is a boolean property, a variable can be sparse or full. We can therefore represent and propagate this attribute as a Boolean lattice as shown in Figure 2.17. The Boolean lattice has four

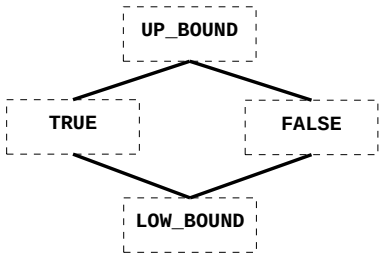

Figure 2.17: MSAD Boolean lattice

elements the default boolean values `TRUE`, `FALSE` and the lattice upper and lower bounds `UP_BOUND`, `LOW_BOUND`. `UP_BOUND` indicates the truth value cannot be determined i.e. it could be either `TRUE` or `FALSE`. `LOW_BOUND` indicates an error condition.

Complexness

Orthogonal to sparse storage, MATLAB allows *complex* arithmetic. An array variable holding complex values will have both a real and imaginary parts. MATLAB determines if the storage class of the result of an operation should be complex depending on the result value. Figure 2.18 shows two examples,

```
1      x = rand(10) + i*rand(10);        x = rand(10) * i*rand(10)
2      y = x * x';                       y = x * x;
3      z = y .* eye(10);                 z = y .* eye(10);

             (1)                               (2)
```

Figure 2.18: Complexness inference example

in `(1)` the second statement computes a matrix product of the complex array `x` and its complex transpose. The variable `y` could be said to be obviously complex because, in this case both operands to the operation on the right hand side (RHS) are complex. However, the result `z` is `real`. To contrast this example consider example `(2)` in which the second statement computes a normal matrix product. In this case the result `z` is complex. MATLAB correctly allocates storage for only real, or both real and complex values in the above examples respectively.

Unless the input values are constants it is not possible to determine if a variable will be complex. Most tools that apply static analysis methods to MATLAB programs conservatively assume that the output will be complex, unless the result is obviously *real* like in the case of `x = zeros(3)` or `s = size(x)`. Like in the case of sparsity a separate Boolean lattice from Figure 2.17 is sufficient to propagate the *complex* attribute.

Although this conservative approach in propagating complexness does not affect MSAD directly (because the target code is MATLAB), for most compiler tools this translates to extra run-time or storage overheads. The generated code needs to determine if any of the arguments are complex and use complex arithmetic operations. The generated code also needs to determine if the result is complex and allocate extra storage to hold the imaginary val-

ues. Both result in run-time overheads. Naively allocating complex storage for all results and performing complex arithmetic on all operations increases both run-time and storage requirement.

Constant Value

In addition to propagating properties of variables listed previously, it is often possible to infer the exact value associated with a variable. This may be from the use of constants or expressions involving constants like `zeros, ones, eye, vertcat, horzcat`. Together with shape inference, value inference may also be able to deduce the value of a shape variable like in the expression `s = size(zeros(4))`. The value of `s` here is $\langle 4, 4 \rangle$.

Definition 2.8 (Constant folding) *Constant-expression evaluation or constant folding refers to the evaluation at compile time of expression whose operands are known to be constant.*

To be able to propagate constants we need to synthesise constant results from expressions with constant operands, in the exact same manner that the target machine or run-time environment would compute, if the compiler would not have synthesised the constant results. This is not very obvious on first thought. The exact results from evaluation constant expressions are highly dependent on the data-type, the machine on which the compiler tool runs also called the *host*, and the machine on which the generated code runs or the *target*. Using double precision floating point arithmetic for example on a machine that does not natively support double precision requires a library that will emulate double precision arithmetic. As long as the standards used by the floating point library and the target machine or run-time environment are the same, the constant folding is valid. There is also the question of overflow and underflow. Different machines handle boundary conditions differently.

Folding boolean and integer arithmetic is relatively straightforward [Muc97], with the exception of handling divide by zero and overflow conditions. Operating on such boundary conditions may cause run-time exceptions, a compiler

therefore has a choice to insert an assertion in the generated code, or generate a compile time warning or an error. Arithmetic operations especially those involving boundary conditions like infinite values `Inf, Inf`, not-a-number `NaN` and significance errors have more serious implications in folding floating-point arithmetic [Gol91]. Commercially used libraries like GNU GMP and MPFR [FHL+07] may be used to perform conforming arithmetic configured for a target on any host machine. Alternately the arithmetic could be folded using the target run-time itself, for example the MATLAB kernel itself could be used to fold constant arithmetic. We use a very simple constant folding library implemented within MSAD.

2.3.2 Control Flow Analysis

So far we have seen how the compiler systematically breaks down a program into an intermediate representation, making it simpler to analyse programs. We have also seen attributes associated with symbols that help organise semantic information gleaned from the program syntax and lexicon, particularly in MATLAB. We also briefly introduced how an inference mechanism can help propagate these attributes through a program, as the result of one operation feeds in to the input of another operation. The programs were however limited to complicated expressions broken down to a sequence of instructions. Figure 2.12 showed how a programming construct like a loop could be simplified into an IR in Figure 2.13. Although the IR is simpler it does not make the dependencies between expressions related through different control paths obvious, which could limit how well the attributes can be inferred. *Control flow analysis* attempts to discover structural properties in a program which helps data-flow analysis, and related compiler optimisations.

Consider the program in Figure 2.19 to efficiently calculate $\mathtt{a}^{\mathtt{n}}\, mod\, \mathtt{z}$ for large integers `a`, `n` and `z`. The function `expmod` uses a `while` loop and an `if` condition. The resuling IR from the lowering phase is shown in Figure 2.20. The first observation from this code is that the loop structure and the condition are no longer obvious. In order to determine a structure in the program we first identify sequences of contiguous statements that execute strictly se-

```
1       function res = expmod(a, n, z)
2           i = n;
3           res = 1;
4           x = mod(a, z);
5           while i > 0
6               if mod(i, 2) == 1
7                   res = mod(res .* x, z);
8               end
9               x = x .^ 2;
10              i = floor(i ./ 2);
11          end
```

Figure 2.19: Function to compute $\mathtt{a}^{\mathtt{n}}$ *mod* $\mathtt{z}$ with loop and condition

rially.

Definition 2.9 (Basic block) *A basic block is a sequence of consecutive statements in which flow of control enters at the begining and leaves at the end without halt or possibility of branching except at the end.*

Algorithm 1, equivalent to the one by Aho et al. [ASU86, Ch.9], demarcates basic blocks given a sequence of contiguous statements. In Figure 2.20 statements `(1) - (3)` belong to block `B0`, `(4) - (5)` belong to `B5`, `(6) - (8)` belong to `B8` and so on. Statements `(11)` and `(16)` are intentionally empty statements and form placeholders for blocks `B7` and `B6` respectively. These surplus blocks are related to MSAD conservatively splitting *critical edges* [CFR+91], which we will re-visit in Chapter 3. The basic blocks are inter-related due to the control flow path linking the blocks of a program in a particular order. A *directed edge* from B_i to B_j indicates if on some or all conditions the sequence of statements in B_j could execute after all the statements in B_i. A graph $G(V, E)$ generated from the basic blocks as the vertices V, and the directed edges between basic blocks E is called a *control flow graph* (CFG). Figure 2.21 shows the control flow graph generated from the IR code in Figure 2.20. Note that in addition to the basic blocks identified from the IR, we have two extra blocks `B3` and `B4`. These are special *entry* and *exit* blocks that are added to every control flow graph. They explicitly

```
1           i = n                          (B0)
2           res = 1                        (B0)
3           x = mod(a, z)                  (B0)
4       L1: var_0 = i > 0                  (B5)
5           if ~var_0 goto L2              (B5)
6           tmp_2 = mod(i, 2)              (B8)
7           var_1 = tmp_2 == 1             (B8)
8           if ~var_1 goto L3              (B8)
9           tmp_3 = res .* x               (B1)
10          res = mod(tmp_3, z)            (B1)
11      L3:                                (B7)
12          x = x .^ 2                     (B2)
13          tmp_4 = i .^ 2                 (B2)
14          i = floor(tmp_4)               (B2)
15          goto L1                        (B2)
16      L2:                                (B6)
```

Figure 2.20: Intermediate form for function `expmod` in Figure 2.19

ALGORITHM 1: Identify basic blocks

Data: S an ordered set of statements in a program
Result: L_B a list of blocks B_i, each an ordered set of statements
begin
  $i \leftarrow 0$, $B_0 \leftarrow \phi$, $L_B \leftarrow \{B_0\}$
  foreach $s \in S$ *in order* **do**
    if *s is a target of a branch* **then**
      if $B_i \neq \phi$ **then**
        $i \leftarrow i + 1$, $B_i \leftarrow \phi$, $L_B \leftarrow L_B \cup \{B_i\}$
      end
      $B_i \leftarrow B_i \cup \{s\}$
    else
      $B_i \leftarrow B_i \cup \{s\}$
      if *s is a branch statement* **then**
        $i \leftarrow i + 1$, $B_i \leftarrow \phi$, $L_B \leftarrow L_B \cup \{B_i\}$
      end
    end
  end
end

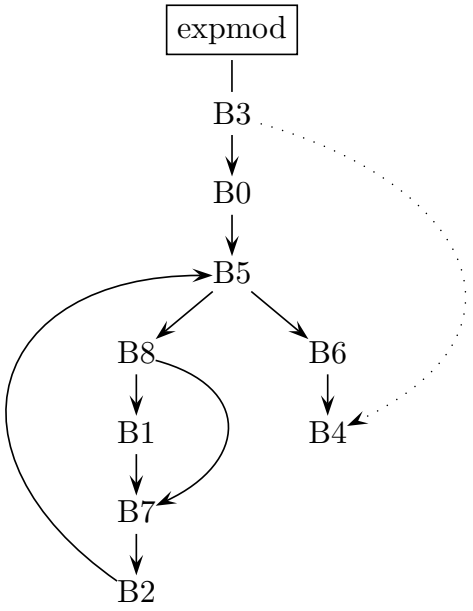

Figure 2.21: Control flow graph for IR code sequence in Figure 2.19

identify the entry and exit points into the control flow graph of a procedure. There is also a virtual edge added from the the entry block to the exit block to simplify further analysis, and does not change the program semantics. The extra edge is again due to a technicality of the IR [CFR+91].

The immediately neighbouring nodes or basic blocks of a block also have special significance. Because the control flow graph is a directed graph i.e. an edge strictly associates one way, the neighbours on the outgoing edges are called *successors*, and on the incoming edges are called *predecessors* [Muc97]. In a control flow graph $G(V, E)$ the successors and predecessors of block b are formally defined by equations (2.8) and (2.9) respectively.

$$Succ(b) = v \in V | \exists e \in E \text{ such that } e = b \rightarrow v \tag{2.8}$$

$$Pred(b) = v \in V | \exists e \in E \text{ such that } e = v \rightarrow b \tag{2.9}$$

The control flow graph forms the basis, directly or indirectly, for all further analyses and optimisations. The next section looks at some useful characteristics that can be gathered from the flow graph that help deal with control

structures.

Dominance

DEFINITION 2.10 (DOMINATOR) *A node* `d` *of a flow graph* ***dominates*** *node* `n`*, written as* `d dom n`*, if every path from the entry node of the flow graph to* `n` *goes through* `d`*. The dominance relationship is both reflexive and transitive.*

If we associate every node with the nodes it dominates, it forms a *tree* also called the *dominator tree*. Figure 2.22 is the dominator tree of the IR code in Figure 2.20. Note that all the basic blocks inside the loop viz.

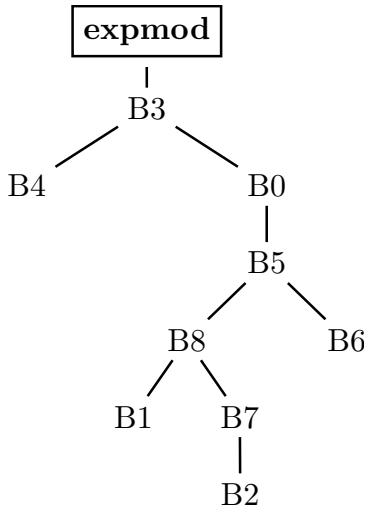

Figure 2.22: Dominance tree for CFG in Figure 2.21

{`B5`, `B6`, `B8`, `B1`, `B7`, `B2`} are dominated by a single block `B5`. This is called the *loop header* and forms a single entry into the loop. We also identify blocks like `B2` in the flow graph, whose head, i.e. `B5`, dominates its tail, `B2`. Such blocks are called the *loop latch* [StG11] and the edge called a *back edge*. All loops have a single header and at least one latch. The MSAD IR also identifies a *loop exit* block `B6` in the case of while loops (and an additional init block in case of for loops). The loop exit block is used to hold some more IR constructs as we will see in Section 2.3.3. The dominance relation is also reflexive (every node dominates itself), transitive (if `a dom b` and `b dom`

`c`, then `a dom c`) and antisymmetric (if `a dom b` and `b dom a` then `b = a`). These properties are useful when testing dominator based block dependencies during various phases.

ALGORITHM 2: Compute dominators

Data: flow graph $G(V, E)$ with set of nodes V, set of edges E and initial node v_0
Result: Dom is a mapping of blocks, v to the set of its dominator blocks, $Dom(v)$

```
begin
    Dom(v_0) ← {v_0}, changed_flag ← true
    foreach v ∈ V − v_0 do Dom(v) ← V
    while changed_flag do
        changed_flag ← false
        foreach v ∈ V − {v_0} do
            T ← V
            foreach p ∈ Pred(v, G) do T ← T ∩ Dom(p)
            D ← {v} ∪ T
            if D ≠ Dom(v) then
                changed_flag ← true
                Dom(v) ← D
            end
        end
    end
end
```

Table 2.1 gives the complete set of dominators for each block of the CFG shown in Figure 2.21 by applying Algorithm 2 [Muc97, Ch.7]. Another useful property of the control flow graphs based on dominance is *postdominance* [FOW87]. We use this property to perform dead code elimination optimisation which we will look at in detail in Section 3.3.1.

DEFINITION 2.11 (POSTDOMINATOR) *A node* **`p`** *of a flow graph* ***postdominates*** *node* **`n`**, *written as* **`p pdom n`**, *if every path from* **`n`** *to the exit includes* **`p`**.

In Section 3.2.3 we will look at an efficient method to compute the dominance relationships. We make use of Corollary 2.12 to re-use the existing

Table 2.1: Dominators using Algorithm 2 applied to CFG in Figure 2.21

b	Dom(b)
B3	{B3}
B4	{B4, B3}
B0	{B0, B3}
B5	{B5, B0, B3}
B8	{B8, B5, B0, B3}
B1	{B1, B8, B5, B0, B3}
B7	{B7, B8, B5, B0, B3}
B2	{B2, B7, B8, B5, B0, B3}
B6	{B6, B5, B0, B3}

efficient dominators implementation to compute the postdominators.

Corollary 2.12
The relationship `p pdom n` *is equivalent to* `n dom p` *in the flow graph if all the edges in the graph were reversed and the entry and exit blocks interchanged [ASU86].*

2.3.3 Data Flow Analysis

The last section introduced control flow analysis as a precursor to analysing programs with control constructs like loops and conditions. Indirectly, the goal of control flow analysis is to help analyse how data variables are related in the presence of changing control flow. *Data-flow analysis* collects global information about how a complex code sequence manipulates associated data. *Information* here implies low level data dependencies that can be used in various code optimisations in the compiler, rather than high level semantics such as an algorithm. The *data-flow information* gathered answers simple questions like, if a variable is used or could be used after a certain point in the program, or which of the several assignments to a variable is relevant at a certain point in the program.

Before looking at the types of analysis and relevant examples, we introduce some terminology that is commonly used in data-flow analysis. Aho et al. [ASU86] describes a *point* as being between two adjacent statements

within a basic block, as well as before the first statement and after the last. And a *path* from point $\mathtt{p}_1$ to $\mathtt{p}_n$ is a sequence of points $\mathtt{p}_1$, $\mathtt{p}_2$, $\cdots$, $\mathtt{p}_n$ such that for each i between 1 and $n-1$, either

1. $\mathtt{p}_i$ is the point immediately preceding a statement and $\mathtt{p}_{i+1}$ is the point immediately following that statement in the same block, or

2. $\mathtt{p}_i$ is the end of some block and $\mathtt{p}_{i+1}$ is the begining of a successor block.

Consider the flow graph in Figure 2.23. Ignoring the exact syntax of the individual statements, basic block `B1` has three points, two before statements `(2)` and `(3)` and one at the end of the basic block after statement `(3)`. There is a path from the start of block `B0` to `B4` leading through all the points in block `B0`, `B1` and `B2`. There is another path leading from `B0` to `B4` through `B3`. We will use the point and path terminology along with lattice operations

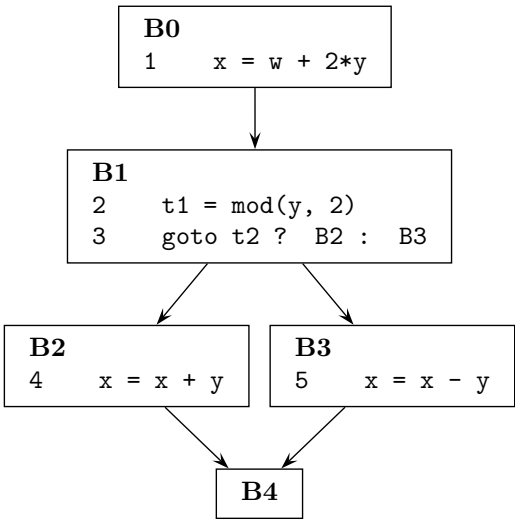

Figure 2.23: Points and path example

introduced earlier to describe data-flow operations in the remainder of this section.

Data-flow analysis is carried out at two levels, local analysis within a basic block, and global analysis at the basic block level across the flow graph. Conventionally the local data-flow information is computed on demand or

iteratively as each statement in the block is processed. The global information is computed and stored in bit-vector lattices making operations on block level data-flow information very efficient.

The exact type of data-flow analysis carried out is based on the kind of information that we seek [Muc97, Ch.8], the direction in which information is propagated, the data-structures and the type of lattices used to hold the information that needs to be collected, etc. The analysis types directly relevant to our work and based on the information collected from a program are:

- *Reaching Definitions* – Determine which definitions of a variable may reach each use of the variable in the program.
- *Live Variables* – Determine for a given variable whether there is a use of this variable along some path from a given point in the program to the exit.
- *Available Expressions* – Determine which expressions are available at each point in a program. The availability of an expression needs to take into consideration that all the variables used in the expression should not be modified between the point at which it is defined up to the point in the program in question along any path between the two points.
- *Copy-Propagation Analysis* – Determine if a variable can be used in place of its copy. Like available expressions, copy-propagation analysis also implies that if a copy operation `x = y` is to be propagated to a use of `x`, variable `y` should be un-modified between the point of the copy operation and the point of the use.

As mentioned earlier data-flow problems can be classified on the direction in which the information is propagated by the analysis. If the direction is along the direction of execution of the program the problem is classed as a *forward problem*, if the direction is opposite to that of execution it is a *backward problem*, or both directions is called a *bi-directional problem*. This classification is also useful because it determines the *data-flow equations*

that must be solved to obtain a solution to the original data-flow problem. The data-flow variables are represented by lattices. Consider the flowgraph $G(V, E)$ with V the set of nodes or basic blocks, including the entry and exit blocks, and E the set of edges connecting the nodes. We associate each basic block $B \in V$ with lattice variables $in(B)$ and $out(B)$ belonging to some lattice $\mathbf{L}$. The variable $in(B)$ represents the data-flow information at entry to basic block B, and $out(B)$ represents the data-flow information at the exit of B. Assuming a forward flow analysis, the data-flow equations can be represented by (2.10), and (2.11).

$$in(B) = \begin{cases} Init & \text{for } B = entry \\ \bigcap\limits_{P \in Pred(B)} out(P) & \text{otherwise} \end{cases} \tag{2.10}$$

$$out(B) = F_B(in(B)) \tag{2.11}$$

Equation (2.10) represents the synthesis of the $in(B)$ lattice value as some combination of the $out(P)$ lattice values of all the predecessor blocks P of block B. The equation (2.11) represents synthesis of the $out(B)$ lattice values as some function $F_B()$ and the data-flow information $in(B)$ at the start of the block B. Note in the presence of loops, or cycles in the CFG, these equations form a recursive set of equations. The implementation of the function $F_B()$ varies according to the type of information we wish to collect from the statements in the block B. The combination operator $\bigcap$ is usually implemented as lattice operations $\wedge$ or $\vee$, again depending on the exact nature of analysis. The lattice element $Init$ is the starting value and is usually set to $\top$ or $\bot$ of lattice $\mathbf{L}$.

Data-flow analysis problems can be solved by many ways such as iterative algorithms, structural analysis, slotwise analysis, interval analysis, syntax-directed approach etc. The two approaches we use are the iterative worklist algorithm [CHK04, Kil73] and *dominance frontier* [CFR+91] based analysis. Iterative algorithms solve a set of data-flow equations until a fixed point is reached. Iterative algorithms have the advantage of handling *irreducible graphs* that occur in the presence of program constructs like `break`

or `continue` [CHK04], present in the MATLAB programming language. Algorithm 3 is an iterative worklist algorithm [Muc97, Ch.8] that solves the forward data-flow equations (2.10) and (2.11).

ALGORITHM 3: Worklist algorithm for iterative data-flow analysis

Data: Flow graph $G(V, E)$ with set of nodes V, set of edges E, $entry \in V$. $Init$ the initial lattice value. And, F the data-flow function

Result: $dfin(v)\ \forall v \in V$ where $dfin(v)$ is the result lattice for node v

```
begin
    dfin(entry) ← Init
    Worklist ← N − {entry}
    foreach v ∈ V do dfin(v) ← ⊥
    while Worklist ≠ φ do
        v ← front(Worklist)
        Worklist ← Worklist − v
        T ← ⊥
        foreach p ∈ Pred(v, G) do T ← T ∩ F(p, dfin(p))
        if dfin(v) ≠ T then
            dfin(v) ← T
            Worklist ← Worklist ∪ Succ(v, G)
        end
    end
end
```

Live Variable Analysis

Live variable analysis determines if a use of a variable exists along some path from a given point in the program to the exit. As we saw in section 2.2.1 the IR generates several intermediate variables each holding the result of evaluating a sub-expression. Once the result of a previous sub-expression is used in a subsequent expression, the old result may not be needed again. For example, consider the IR code in Figure 2.20. For clarity we annotate the CFG in Figure 2.21 with the IR as shown in Figure 2.24. The variables `tmp_2`, `tmp_3` and `tmp_4` are both defined and used in basic blocks `B8`, `B1` and `B2` respectively. Variable `tmp_2` is said to be *live* at the point between

statements (6)-(7). The fact that some variables cease to be used may also be true of program variables and not just those created by lowering to IR. For example in Figure 2.20, the variables a and n are only used in block B0. Variable a is an argument and is live between all the points in the entry block B3 up to statement (3). The only variable live at and after statement (16) is res the result variable, all other variable cease to be live past this point.

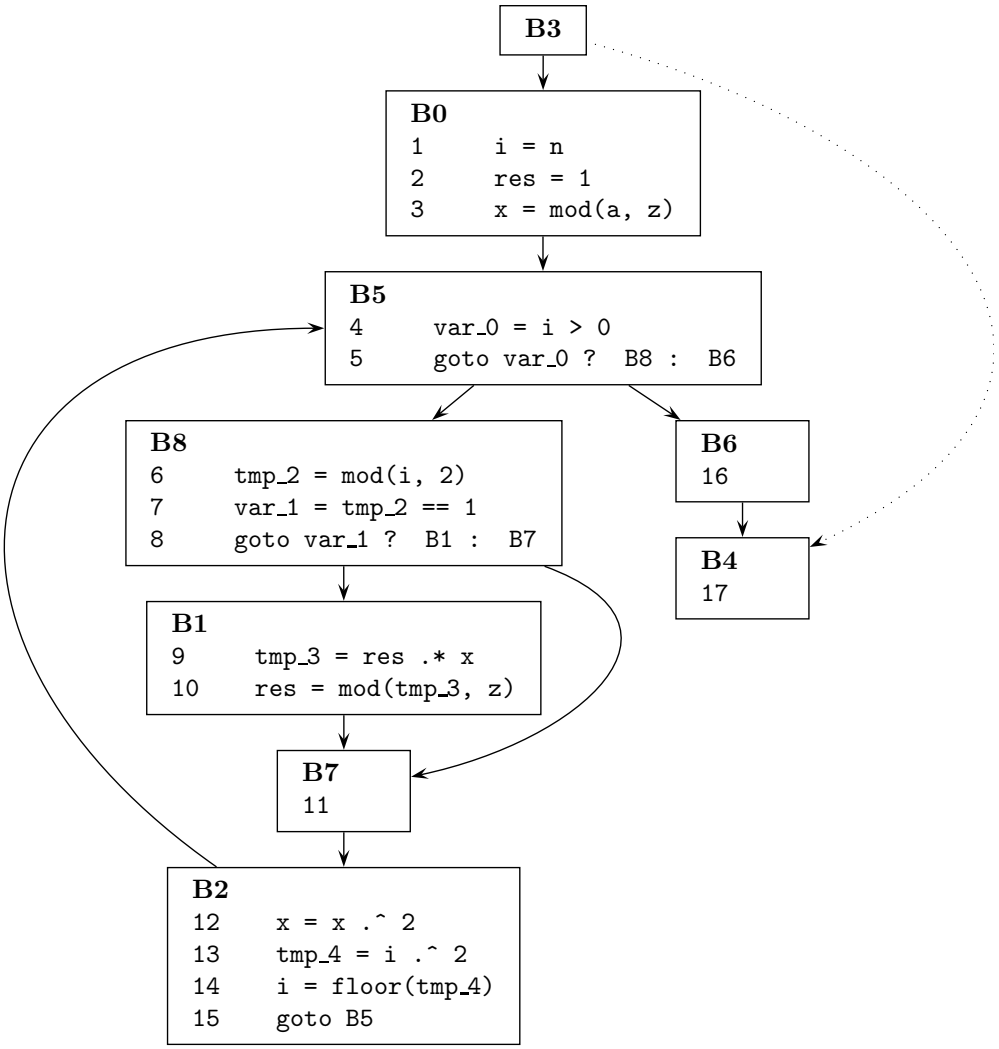

Figure 2.24: CFG in Figure 2.21 with IR code

The most common use of live variable information is during code generation. When generating machine code, all the variables (pseudo-registers)

in the IR eventually need to be mapped to physical registers on the target processor. All machines have a limited set of registers that can be used for operations, which implies the registers need to be re-used. Identifying live variables is the first step to determining which intermediate results may be expunged to hold a result from evaluating another expression. We use liveness information to reduce the number of SSA temporaries created to cope with change in control flow, and to coalesce copies of composite variables like arrays, structures and cells, discussed later in Chapter 3.

Formally live variable analysis is a backward flow problem as the uses of variables need to be propagated in reverse to the control flow. Analogous to the data-flow equations (2.10, 2.11) for forward flow, equations (2.12, 2.13) represent the backward flow of information. Note that the flow function F_B is well defined in this case and computes the liveness information [ASU86, Ch.10].

$$out(B) = \begin{cases} Init & \text{for } B = exit \\ \bigvee_{S \in Succ(B)} in(S) & \text{otherwise} \end{cases} \tag{2.12}$$

$$in(B) = use(B) \vee (out(B) - def(B)) \tag{2.13}$$

The lattice values $use(B)$ and $def(B)$ form inputs to these equations and represent the lattice values corresponding to the use and the definition of a variable in block B. Because we wish to determine if a variable is live along any path, and not necessarily all paths from a point to the exit, the lattice join operation $\vee$, models the combining of successor liveness information as we traverse the CFG in reverse. The appropriate value for $Init$ here is $\perp$ to indicate no variable is live at the end of the exit block. The result variables are assumed to be implicitly used in the exit block making them live at the start of the exit block. The propagation of liveness information is carried out by the iterative scheme described in Algorithm 3. As is, the algorithm solves only forward data-flow problems, but it can be easily adapted to solve backward data-flow problems like live variable analysis by simply substituting variables $dfout$ for $dfin$, $exit$ for $entry$ and $Succ$ for $Pred$. MSAD's implementation of

Algorithm 3 is parameterised such that it can be used to solve either forward or backward data-flow problems.

If each variable is represented by a single bit in a bit-vector associated with every block, Table 2.2 shows the result of live variable analysis applied to the complete code in Figure 2.24. The *use* and def sets are pre-computed and form inputs to the iterative algorithm. The following template shows the position in the bit-vector for each variable in the IR of Figure 2.24:

$$\langle tmp4,\ tmp3,\ var1,\ tmp2,\ var0,\ x,\ res,\ i,\ z,\ n,\ a \rangle$$

Table 2.2: Live variable analysis applied to the CFG in Figure 2.24

block	Use(b)	Def(b)	LvOut(b)
B3	⟨00000000000⟩	⟨00000000111⟩	⟨000000010111⟩
B4	⟨00000010000⟩	⟨00000000000⟩	⟨000000000000⟩
B0	⟨00000000111⟩	⟨00000111000⟩	⟨000000111100⟩
B5	⟨00000001000⟩	⟨00001000000⟩	⟨000000111100⟩
B8	⟨00000001000⟩	⟨00110000000⟩	⟨000000111100⟩
B1	⟨00000110100⟩	⟨01000000000⟩	⟨000000111100⟩
B7	⟨00000000000⟩	⟨00000000000⟩	⟨000000111100⟩
B2	⟨00000101000⟩	⟨10000000000⟩	⟨000000111100⟩
B6	⟨00000000000⟩	⟨00000000000⟩	⟨000000010000⟩

Use-Def and Def-Use Chains

Def-use chains, or simply *du-chains* are a sparse representation of the *reaching definitions* data-flow information that we introduced earlier. Du-chains connect a definition of a variable to all the uses that the definition may reach. Similarly use-def chains, or *ud-chains*, are a sparse representation of the *upward exposed uses* data-flow information. The *upward exposed uses* data-flow analysis is the dual of reaching definitions [Muc97]. Where reaching definitions is a forward problem propagating available definitions towards uses, upward exposed uses is a backward problem that propagates uses in reverse towards definitions. A ud-chain associates a use of a variable to all the

definitions that may reach that use.

Consider the IR in Figure 2.24. If the definitions and uses of variables are denoted by the block-statement pair in which they occurs, the result variable `res` has two definitions `(B0,2)` and `(B1,10)`. Variable `res` has two uses `(B1,9)` and another implicit use `(B4,17)` in the exit block `B4`. Because `B1` is inside a loop, both definitions of variable `res`, `(B0,2)` the initial value outside the loop and `(B1,10)` the recursive definition inside the loop, can reach the use of `res` in `B1`. Also, whether the loop body is evaluated depends on the value of `i` implying both definitions of `res` can reach the implicit use in `B4`. The du-chain for definition `(B0,2)` thus includes both uses `(B1,9)` and `(B4,17)`. Similarly the du-chain for definition `(B1,10)` includes both uses `(B1,9)` and `(B4,17)`. In this case the ud-chains are similar with each use `(B1,10)` and `(B4,17)` containing both the definitions `(B0,2)` and `(B1,10)` each. The definitions for the two uses are said to be ambiguous because we cannot statically determine exactly which unique definition is relevant at the use of the variable. Trivially the use `(B8,7)` of variable `tmp_2` for example, has an unambiguous definition `(B8,6)`.

MSAD uses a *static single assignment* (SSA) form of the IR which we introduce in the following section. The SSA form makes the du-chains explicit i.e. multiple definitions reaching each a use are dis-ambiguated by creating a unique pseudo-variable corresponding to every definition of the variable. SSA improves the effectiveness of several code optimisations which we will introduce in Section 2.3.4.

Static Single Assignment (SSA)

In this section we introduce the SSA [CFR+91] form of intermediate representation and its relevance to MSAD. The translation in and out of SSA form is described in detail in Chapter 3. Section 2.2.1 emphasised the need for a simpler intermediate representation of the program source. Sections 2.3.2 and 2.3.3 demonstrated some of the techniques by which the IR can be analysed and useful control- and data-flow information extracted. Because code optimisations heavily rely on control- and data-flow information, the effec-

tiveness of any code optimisation is as good as the quality of the flow information. The previous section described the concept of du- and ud-chains to identify related definitions and uses. The presence of ambiguity in the exact definition that reaches a use increases the complexity in analysis, and can impact application of optimisations such as constant propagation, redundancy elimination, invariance detection in loop optimisations, and global value numbering [CCF91]. Essentially the SSA form solves the problem of ambiguous definitions by providing a unique name to every assignment of the same variable in a procedure, and all the uses reached by that assignment are renamed to the new variable name. This straightforwardly applies to straight line code or branches, however where two or more control flow paths meet, multiple definitions may be required to be fused together. The fusing of copies is done using special ϕ constructs which we will see later in this section. Once the procedure is in the SSA form, the du-chains are explicit i.e. there is a single definition that reaches all the uses of that variable in the procedure. Data-flow analysis of programs in the SSA form have the following benefits [CCF91]:

1. Information is combined as early as possible.

2. Information is forwarded directly to where it is needed.

3. Useless information is not represented.

As described in the previous paragraph, SSA creates a unique copy of a variable for every (re-)definition of the same variable in a procedure. To make this copy unique, the variable name is simply suffixed by a number. The suffix number is allocated per assignment to the same variable, typically the next number in increasing order. If a variable has independent assignments along two branches of a condition, each definition will receive a new variable. Although in a straight line code, each new definition of the same variable naturally renders the previous definition *not-live*, in the case of a branch both definitions are live where the two paths meet. For example, in Figure 2.24 the two definitions `(B0,3)` and `(B2,12)` of the variable `x` overlap in block `B5`, and are subsequently used in `B1` and `B2`. After converting to SSA the

two definitions will create copies of variable `x`, say `x_1` and `x_3` in blocks `B0` and `B2` respectively. The two definitions overlap in block `B5`, also called the *join*. Because variable `x` is then used in `B1` and `B2` both dominated by block `B5`, both the definitions could reach the two uses. We therefore need to merge the two copies into a single variable. Merging of copies is achieved by the so-called ϕ assignments. The augmented CFG and IR code

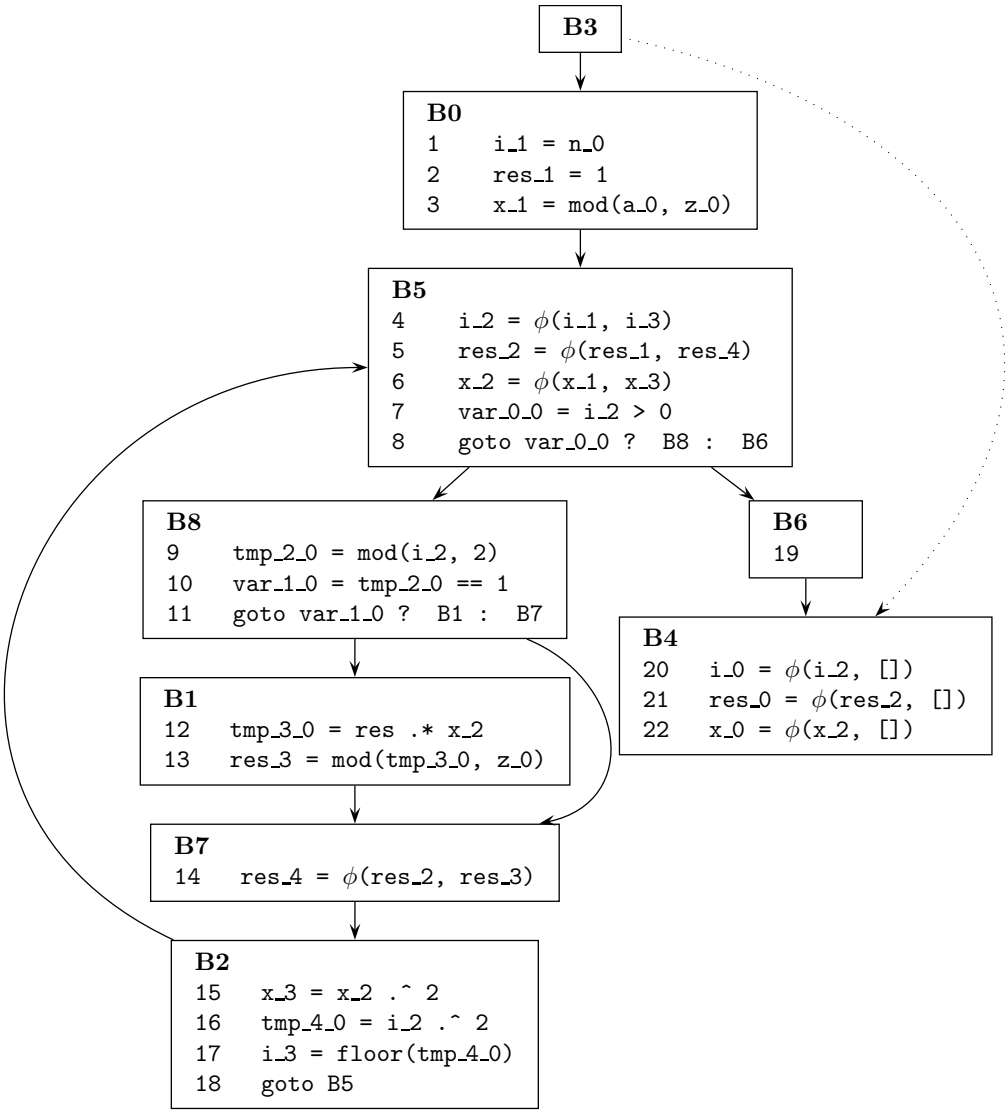

Figure 2.25: SSA form of IR code in Figure 2.24

in Figure 2.25 represents the SSA IR form of the IR code in Figure 2.24. We observe the ϕ nodes placed in block `B5` merge copies of the variable `x`, and other multiply defined variables `i` and `res`. Note each variable has a suffix that matches the pattern `_[0-9]+`. The number indicates the n^{th} assignment to the variable. Also, implicitly associated with each argument to a ϕ is the immediate predecessor block along the path that provides that definition. For example, argument `res_3` in statement `(B7,14)` is associated with block `B1`, and `res_2` is associated with `B5`.

The trivial ϕ nodes in exit block `B4` result from the virtual edge between the entry and the exit block, and are inconsequential. The variables `i`, `res` and `x` are not arguments to the function, neither are they defined in `B3`. The definitions for these variables reaching `B4` from `B3` are therefore denoted by the empty notation `[]`.

Other than simplifying def-use analysis, dominance based analysis and optimisations mentioned earlier in the section, the SSA form is also very relevant to processing MATLAB. In Section 1.4 we talked about MATLAB as a dynamically typed language i.e. the type of a variable is determined by the value that defines it. If a MATLAB variable is re-defined, the variable may change shape, size and sparsity if it is an array variable, and even the type. The SSA form lends itself to this model as every definition is associated with a new variable. It therefore becomes straighforward to assign, analyse and propagate the new properties of a variable following a re-definition of the variable.

2.3.4 Optimisations

In the context of a compiler, the ultimate goal of all control- and data-flow analysis described so far is to be able to optimise and transform the input program to machine code. In the context of MSAD the goal is to specialize the input program with respect to overloaded AD operations, apply generic optimisations, and generate an output that will directly or indirectly be executable. Generally, specialisation of programs can be with respect to any of the properties identified to be revelant to an application, like those

in Section 2.3.1 relevant to MSAD. In compiler terminology specialisation of various properties translates to applying relevant optimisations, some of which we will briefly look at in this section. The algorithms and implementation details will be covered in Chapter 3. In general, code optimisation is a superset of specialisation and includes code improving transformation above and beyond the scope of specialisation alone.

Constant Propagation

Constant propagation optimisation discovers variable values that are **always** constant over all paths of execution from the definition to the use, and replaces relevant uses of these variable by the equivalent unique constant value. There are two types of constant propagation methods, local and global. The local variant is concerned with propagating constants only within a basic block. The more interesting *global constant propagation* on the other hand propagates constants across basic blocks taking into account control flow. To ensure the propagated value is truly constant across all paths leading to the use, propagation algorithms are usually iterative and conservative.

Constant propagation is an optimisation because it very often results in sub-expressions with all constant operands *folded* at compile-time reducing the run-time overhead of evaluating the constant expression. Constants can also be beneficially propagated into predicates of conditional statements. If the predicate is evaluated to be `true` or `false` at compile time, other optimisation passes such as dead code elimination or branch optimisations may be able to simplify or remove the branching and conditional execution. Because executing a branch instruction on most processors implies a performance penalty both for the CPU and the instruction cache, removing redundant branches is highly desirable

Consider the example in Figure 2.26. The function generates a tridiagonal matrix using input `x`. Although there is no obvious opportunity to apply constant propagation to this code as is, it is often possible to optimise a callee function in the context of the caller function that supplies the arguments at the function call site. If function `const_prop` is called with

```
1       function y = const_prop(x, m)
2           n = size(x, 1);
3           if n < 128;
4               y = zeros(n);
5           else
6               y = spalloc(n,n,3*n);
7           end
8           for i = 1:n
9               if i > 1
10                  y(i-1,i) = 2 .* pi ./ m .* x(i-1,1);
11              end
12              y(i,i) = 2 .* pi ./ m .* x(i,2);
13              if i < n
14                  y(i+1,i) = 2 .* pi ./ m .* x(i+1,3);
15              end
16          end
```

Figure 2.26: Function to create a tri-diagonal matrix using `x`

```
1           n_0 = size(x_0, 1)
2           var_0 = n < 128
3           if ~var_0 goto L1
4           y_1 = zeros(n_0)
5           goto L2
6       L1: tmp_0 = 3 * n_0
7           y_0 = spalloc(n_0, n_0, tmp_0)
8       L2: y_2 = phi(y_1, y_0)
```

Figure 2.27: IR for statements `(2) - (7)` of function `const_prop` in Figure 2.26

arguments `x` which is known to be a matrix with shape $\langle 16, 3 \rangle$, and `m = 16`, we can propagate this partial information as constants through the program. Figure 2.27 shows the intermediate representation for a subset of statements `(2)-(7)` of function `const_prop`. If the constant expression `size(x,1)` is evaluated at compile time to be `16`, we can replace all uses of the variable `n` by `16`. Figure 2.28 shows the replacements made by constant propagation. Note that in addition to simply propagating constant values, constant folding has simplified expressions involving constant operands. The value of `var_0` in statement `(2)` has been simplified to be `1` (for clarity the original constant expression is shown as a comment). Similarly constant expressions in statements `(3)` and `(6)` can be simplified to `0` and `48` respectively. Typically constant folding, if applicable, is carried out at every step of constant propagation, which allows constant expressions to be simplified early and information propagated as far into the program as possible.

```
1           n_0 = 16
2           var_0 = 1                       % (16 < 128)
3           if 0 goto L1                    % (~1)
4           y_1 = zeros(16)
5           goto L2
6       L1: tmp_0 = 48                      % (3 * 16)
7           y_0 = spalloc(16, 16, 48)
8       L2: y_2 = phi(y_1, y_0)
```

Figure 2.28: Specialised IR corresponding to IR in Figure 2.27 for `size(x) = [16, 3]` by applying constant propagation and constant folding

Looking at statement `(3)` in the optimised IR in Figure 2.28 we observe that the conditional branch has a constant predicate that implies the statement will always take the same branch to statement `(4)` irrespective of any state of the program variables. In the following section we discuss *dead code elimination* which can optimise away unreachable code like statements `(6)-(7)` in Figure 2.28. Figure 2.29 shows the overall impact of these optimisations after converting the optimised code back from IR to MAT-

LAB. It is equivalent to function `const_prop` in Figure 2.26 but *specialised* for `size(x) = [16, 3]` and `m = 16`. This resulting specialised function `const_prop_s16x3_v16` is also called the *residual* in context of *partial evaluation*. Note the redundant `else` part of the `if-else` construct, statement `(6)` in Figure 2.26, has been removed. The constant expressions `2 * pi / m` has been simplified to a single constant value in statements `(5)`, `(7)` and `(9)`.

```
1       function y = const_prop_s16x3_v16(x, m)
2           y = zeros(16);
3           for i = 1:16
4               if i > 1
5                   y(i-1,i) = 0.3926990816987241 .* x(i-1,1);
6               end
7               y(i,i) = 0.3926990816987241 .* x(i,2);
8               if i < 16
9                   y(i+1,i) = 0.3926990816987241 .* x(i+1,3);
10              end
11          end
```

Figure 2.29: Function in Figure 2.26 specialised for `size(x) = [16, 3]` and `m = 16`

MSAD implements a very efficient and accurate SSA IR based *Sparse conditional constant propagation* (SCCP) [WZ91] algorithm to propagate the augmented lattice of all properties previously identified in Section 2.3.1. The implementation details are discussed later in Section 3.3.2. The *value* property of variables identified earlier, together with a boolean lattice effectively forms a constant lattice, which is propagated using the SCCP algorithm.

Dead Code Elimination

As part of specialising a program one of the more important optimisations, after constant propagation, is *dead code elimination* (DCE). Program statements that have side-effects i.e. direct or indirect assignment to variables, are *live* if the variables assigned to are used elsewhere in the program, otherwise

the statement is *dead*. As the name of the optimisation suggests, removal of code that is *not-live* or *dead*, is termed as dead code elimination. We also class *unreachable code* that is never executed by virtue of the control flow or data-invariants, as dead code. Although DCE applies even to input programs, dead code is seldom found in user coded programs. The most significant impact of this optimisation is in the intermediate stages of optimisation where other optimisations or transformations might expose dead code.

In the previous example in Figure 2.28, forward propagation of constants leaves assignment statements `(1)` and `(2)` dead because variables `n_0` and `var_0` have no uses. Constant propagation also leaves statements `(6)` and `(7)` obviously unreachable because the branch in statement `(3)` is never taken. After removing all the dead statements above, we get the optimised IR code in Figure 2.30.

```
1           y_1 = zeros(16)
2           goto L2
3       L2: y_2 = phi(y_1, [])
```

Figure 2.30: Dead code elimination applied to IR in Figure 2.28

Applying constant propagation followed by DCE to the complete program in Figure 2.26 and converting the IR back to MATLAB, we get the optimised program in Figure 2.29. This specialised function is smaller in size, has one less condition test and branch due to constant propagation and DCE, and has three floating point multiplies and three divide operation inside the loop optimised away due to constant propagation and folding.

We use a number of methods to implement DCE which are dealt with in detail in Chapter 3. We use an algorithm similar to that described by Muchnick [Muc97, Ch.18] to remove obviously unreachable code in the IR. In order to analyse def-use dependencies across basic blocks and delete dead code based on liveness we use Cytron's *control dependents based dead code elimination* [CFR+91]. Branch optimisation triggered by constant predicates

can cause sections of the CFG to be optimised out. We use the dominator tree to identify all child blocks of a parent that have become unreachable to remove unreachable code. Finally, trivially dead code which computes results that have no uses left are removed as they become redundant in the relevant optimisation phase. We record uses for every definition in explicit use-def chains in order to delete dead code.

Function Inlining

Function Inlining or *procedure integration* replaces calls to functions by the body of the called function at the point of call. The called function is termed the *callee*, the function in which the callee is called is the *caller*, and the point at which the caller calls the callee is the *call-site*.

```
1       function [z, ad_z] = ad_test_inline(x, ad_x)
2           y = rand(10,1);
3           ad_y = rand(10, size(x, 2));
4           [z, ad_z] = ad_plus(x, ad_x, y, ad_y);
5
6
7       function [c, d_c] = ad_plus(a, d_a, b, d_b)
8           c = a + b;
9           ssa = numel(a);
10          ssb = numel(b);
11          if ssa == ssb
12              d_c = d_a + d_b;
13          elseif ssa == 1
14              d_c = d_a(ones(1, ssb), :)  + d_b;
15          elseif ssb == 1
16              d_c = d_a + d_b(ones(1, ssa), :);
17          end
```

Figure 2.31: Optimisation by function inlining

Consider the program in Figure 2.31. Ignoring the purpose of the program at this point, we observe that function `ad_test_inline` calls function `ad_plus` in statement `(4)` with four arguments. As described earlier, function inlining replaces the call, in this case statement `(2)` with the callee

function body, `ad_plus`. To replace the function call and preserve the original caller function semantics we first need to associate the arguments in the function call, *actual arguments*, with the callee's arguments in the function call declaration, *formal arguments*. We also need to map the result variables in the callee function to the result variables in the caller function. If any of the variables in the callee function share the same names as variables in the caller function, they need to be renamed to avoid interfering with the original caller function variables. Figure 2.32 shows the result of inlining the function `ad_inline`.

```
1       function [z_0, ad_z_0] = ad_test_inline(x_0, ad_x_0)
2           y_0 = rand(10, 1);
3           ad_y_0 = rand(10, size(ad_x_0, 2));
4           a_0 = x_0;
5           d_a_0 = ad_x_0;
6           b_0 = y_0;
7           d_b_0 = ad_y_0;
8           c_1 = a_0 + b_0;
9           ssa_0 = numel(a_0);
10          if ssa_0 == 10
11              d_c_4 = d_a_0 + d_b_0;
12          elseif ssa_0 == 1
13              d_c_4 = d_a_0(ones(1, 10), :)  + d_b_0;
14          end
15          z_0 = c_1;
16          ad_z_0 = d_c_4;
```

Figure 2.32: Program in Figure 2.31 after inlining (and applying constant propagation and dead code elimination optimisations)

In the resulting code in Figure 2.32 we can see the interface copies assigning the actual arguments to the formal arguments in statements `(4-7)`, and the results copied out in statements `(15-16)`. These copy statements map the actual and formal arguments. In this case there are no shared names between the caller and the callee. The last two arguments `y` and `ad_y` in the call to `ad_plus` are setup by the caller function `ad_test_inline`. The MATLAB builtin function `rand` creates an array of the specified dimensions and pop-

ulates the elements with random numbers. The size of `y_0` can therefore be inferred to be $\langle 10, 1 \rangle$. Constant propagation evaluates `ssb` to be `10` and forward propagates this value to statements `(11)` and `(15)` in Figure 2.31. The predicate in statement `(15)` evaluates to false making statements `(15-16)` redundant which are removed by DCE along with statement `(10)`. Together with function inlining, Figure 2.32 demonstrates the application of constant propagation and DCE described earlier. However, in the optimised code we observe several copy statements (interface copies) generated as a result of inlining. In the following section we introduce the copy propagation optimisation which removes any redundant copies.

In Section 3.3.3 we will look at the implementation of function inlining in MSAD in detail. As described in Section 1.3.1, inlining of functions needs careful consideration, but has distinct advantages. According to Muchnick [Muc97, Ch.15] some of the factors that may be used to control inlining are:

1. Size of the procedure body and estimate of decrease in size after inlining
2. The number of calls to the callee procedure
3. Whether the callee procedure is called inside a loop
4. Whether the call includes one or more constant-valued parameters.

Function inlining is generally classified on the basis of the stage at which the optimisation is applied. The most common is *early inlining* which uses heuristics based on factors mentioned above to decide which function calls to inline before any of the code optimisations are applied. MSAD however uses a *late inlining* approach and defers inlining functions until many of the optimisations we have described earlier have been applied. Although this complicates the process of inlining with additional effort to merge the control- and data-flow information, it has the following advantages in the context of MSAD:

1. Late inlining allows constant propagation to propagate the type lattice of a variable before attempting to resolve the overloaded method. This

is vital for the purpose of AD where we wish to resolve and specialise the overloaded operations provided by the MAD package.

2. Along with the type, attributes like size, rank and other properties described in Section 2.3.1 are propagated in advance which helps specialisation of the overloaded function.

3. Late inlining also reduces the bulk of the IR because dead code is eliminated as early as possible thereby eliminating the additional complexity to process statements which do not affect the result of the function.

Copy Propagation

Copy propagation replaces uses of a copy variable `x` created by assignments of the form `x = y`, with the copied variable `y` provided that neither variable change value along all the paths that lead from the definition of `x` to the particular use. Similar to constant propagation there are two types, *local copy propagation* that performs copy propagation within a basic block, and *global copy propagation* operates across basic blocks.

Variable copies may not occur in user code, but may be generated in the IR by various intermediate optimisations like common subexpression elimination or induction-variable optimisation [ASU86, Ch.10]. In MSAD, function inlining or removing SSA generated ϕ nodes may also insert variable copies. Although copy operations may not appear to impact performance directly, they increase code size especially revelant in small loops, and they may hinder some optimisation that do not look through copy operations i.e. track equivalence of copy variables.

Comparing the local copy propagation optimised function in Figure 2.33 and the un-optimised function in Figure 2.32 we can see the copy assignment statements `(4)` and `(6)` in Figure 2.32 have been removed by forwarding variables `x_0` and `y_0` to the uses of their respective copy variables `a_0` and `b_0` in statements `(8)` and `(9)` in the same basic block. We also observe that copy statements `(4 - 5)` and `(13 - 14)` in Figure 2.33 still remain because the definition and use of variables `d_a_0`, `d_b_0`, `z_0` and `ad_z_0` span across

```
1       function [z_0, ad_z_1] = ad_test_inline(x_0, ad_x_0)
2           y_0 = rand(10, 1);
3           ad_y_0 = rand(10, size(ad_x_0, 2));
4           d_a_0 = ad_x_0;
5           d_b_0 = ad_y_0;
6           c_1 = x_0 + y_0;
7           ssa_0 = numel(x_0);
8           if ssa_0 == 10
9               d_c_4 = d_a_0 + d_b_0;
10          elseif ssa_0 == 1
11              d_c_4 = d_a_0(ones(1, 10), :)  + d_b_0;
12          end
13          z_0 = c_1;
14          ad_z_0 = d_c_4;
```

Figure 2.33: Inlined function in Figure 2.32 after local copy propagation

```
1       function [z_0, ad_z_1] = ad_test_inline(x_0, ad_x_0)
2           y_0 = rand(10, 1);
3           ad_y_0 = rand(10, size(ad_x_0, 2));
4           c_0 = x_0 + y_0;
5           ssa_0 = numel(x_0);
6           if ssa_0 == 10
7               d_c_4 = ad_x_0 + ad_y_0;
8           elseif ssa_0 == 1
9               d_c_4 = ad_x_0(ones(1, 10), :)  + ad_y_0;
10          end
11          z_0 = c_1;
12          ad_z_0 = d_c_4;
```

Figure 2.34: Inlined function in Figure 2.32 after local and global copy propagation

basic blocks. MSAD also applies a global copy propagation optimisation based on Use-Def chains to propagate single def copies across basic blocks. In the example in Figure 2.33 variables `d_a_0` and `ad_x_0` both linked through copy statement `(4)`, have a single definition each in statements `(1)` and `(4)` respectively. This implies all uses of the variable `d_a_0` dominated by the definition in statement `(4)` unambiguously refer to this definition, and can be replaced by the variable `ad_x_0`. Note that the definition of `ad_x_0` implicitly dominates definition of `d_a_0`. In Figure 2.34 we observe global copy propagation has forward propagated copies in statements `(4)` and `(5)` from Figure 2.33. The only copies remaining are statements `(11 - 12)` in Figure 2.34. These copies are the side-effect of removing *phi* nodes which we will deal with in Chapter 3 where we demonstrate the use of SSA copy coalescing which allows us to further remove such copies.

Algorithm 4 describes local copy propagation optimisation, as demonstrated above, and is based on the algorithm in Muchnick [Muc97, p.357]. The algorithm iterates through every statement in a basic block replacing variables on the RHS (right hand side) of an expression with replacement copy variables queried from a replacement map. The map stores paired copy variables $\langle lhs, rhs \rangle$ which are added when a copy statement is encountered. The replacement is also updated every time an assignment statement is encountered where all the variables on the LHS (left hand side) are removed. The supporting routines in Function FindCopyReplacement and Function RemoveReplacement add and remove copy pairs from the replacement map.

Function RemoveReplacement(CopyMap, v) update CopyMap

```
begin
    foreach set element cme ∈ CopyMap do
        if v = cme.lhs ∨ v = cme.rhs then
            CopyMap ← CopyMap − {cme}
        end
    end
end
```

ALGORITHM 4: Local forward copy propagation

Data: Basic block B_i which to apply local copy propagation to
Result: B_i with statement $S_k : z \leftarrow f(x) \Rightarrow S_k : z \leftarrow f(y)$, iff $\exists_{j<k} S_j : x \leftarrow y, \nexists S_m : x \leftarrow g(), \nexists S_n : y \leftarrow h() \; \forall \; j < m, n < k$

```
begin
    CopyMap ← φ          %set of paired elements <lhs,rhs>
    foreach statement S ∈ B_i in sequence from start to end do
        foreach variable v used in statement S do
            rep ← FindCopyReplacement(CopyMap, v)
            if rep ≠ φ then
                Substitute rep in place of v in statement S
            end
        end
        if S is an assignment statement then
            foreach variable l defined in statement S do
                RemoveReplacement(CopyMap, l)
            end
        end
        if S is a copy assignment statement, lvar = rvar then
            Add pair ⟨lvar, rvar⟩ to CopyMap
        end
    end
end
```

Function FindCopyReplacement(CopyMap, v) returns rep

```
begin
    rep ← φ
    foreach set element cme ∈ CopyMap do
        if v = cme.lhs then
            rep ← cme.rhs
        end
    end
    return rep
end
```

Chapter 3

Implementation

Chapter 1 covered the basics of AD and some of the issues with performance of MATLAB programs, especially relevant in the context of AD. We also discussed how compiler optimisations could tackle these performance issues in a source transformation framework for AD. In Section 1.5 we proposed that the problem of applying AD to a given program can be reduced to a compiler problem of resolving the overloaded operations of the MAD package classes `fmad` and `derivvec`. Chapter 2 covered the basics of compiler techniques relevant to MSAD, and the minimal framework required to implement source transformation of programs. This chapter focuses on MSAD and covers the implementation details and algorithms used in MSAD. It also demonstrates how specialisation is used to generate source transformed AD code from overloaded operations. The following text makes generous use of the compiler terminology introduced earlier in Chapter 2 and therefore assumes the reader has read Chapter 2 or is generally aware of compiler terminology.

Much like the compiler structure introduced in Chapter 2, MSAD operates in phases or passes. Each pass systematically translates, transforms or adds information to the result of the previous pass. MSAD uses its own medium level intermediate representation (MIR) to internally represent MATLAB code. All high level MATLAB data structure constructs like array accesses, structure field accesses, etc. and control constructs like `if-elseif-else`, `for`, `while` etc. are reduced to a simple lower level forms amenable to analysis

and transformations. This simpler MIR avoids MSAD having to deal with complicated semantics of high level data- and control-flow structures and manipulating complex MATLAB syntax. The details of the MIR itself are dealt with in Section 3.1.5.

Section 3.4 covers the process of augmenting operations in the source program with AD operations. In MSAD this augmenting operation is carried out by specialising and inlining the overloaded operations from MAD in a separate pass. If we choose not to apply this pass, MSAD simply serves as a MATLAB source to source optimisation platform. The sections leading to Section 3.4 delve into details of the MIR in MSAD, generating the MIR, passes to carry out control-flow and data-flow analysis introduced earlier in Section 2.3 on the MIR, transformation and optimisation passes and finally converting the MIR back to executable MATLAB code.

3.1 Lowering

As described earlier in Section 2.2.1 an optimising compiler internally operates on a simpler intermediate representation, as compared to the complicated semantics and syntax of MATLAB in the case of MSAD. The process of converting the high-level language of the supplied input program to a lower level IR is called *lowering*. This section will cover details of parsing MATLAB and generating the MIR.

3.1.1 ANTLR support

In Chapter 2 we have seen some of the compiler techniques that go into building a language processing tool and we can appreciate the complexity involved in implementing even basic operations like scanning and parsing. For most applications a more practical approach, is to make use of existing tools like Lex [LS] and Yacc [Joh75], ANTLR [PQ95] and JavaCC [Sun] that generate parsers from grammar specifications, which may be used for the parsing requirements within the larger framework of these application. These tools have been used in numerous small and large applications like

compilers, interpreters, text formatters, structure editors, query interpreters, etc. Octave [Eat11] a MATLAB clone, for instance uses a Lex and Yacc generated parser to parse the script file for execution [Eat11]. We make use of ANTLR generated parsers in MSAD. ANTLR has also been used in the EliAD [FTPR04] project for elimination AD applied to Fortran codes.

ANTLR short for Another Tool for Language Recognition, is a predicated LL(k) parser generator, that generates code for lexical and syntax analysers, and tree walkers [PLW+04]. Until recently most tools generated the conventional DFA based lexers and LALR based parsers, whereas ANTLR 2.7.7 uses LL(k) grammars for generating both the scanner and parser. In addition to the support for scanning and parsing, ANTLR also facilitates the generation of an AST, and builds tree walkers for traversal of the AST with supporting actions being carried out at each node. These actions can be used to execute code to perform symbol table management, attribute synthesis, tree transformation and code generation.

Reasons for using ANTLR

ANTLR provides several features that make its use, in this implementation, favourable. Among other features, the following have been most beneficial:

1. ANTLR comes with automatic AST generation and an integrated AST parser generator which is useful in the post-parsing phases. Moreover, the grammar for this is similar to the original EBNF parser grammar, making its implementation and debugging easier.

2. As compared to other tools, the default run-time error handling with ANTLR is much better. It is also possible to extend these capabilities further with targeted exception handling.

3. ANTLR generates a recursive descent parser very similar to hand built parsers, this simplifies any debugging effort, and if required allows painless tweaking of the generated parser.

4. ANTLR provides predicates which lets the programmer systematically direct parsing via arbitrary expressions using semantic and syntactic

context. This also eliminates the need to hand-tweak the ANTLR output, even for difficult parsing problems [PQ95].

5. ANTLR generated scanners also use LL(k) grammars, and the code generated using this strategy is supposed to be much faster than the standard DFA based lexers [PQ95].

3.1.2 Scanning

The scanning pass in MSAD performs lexical analysis of the input character stream (input file) and groups specified sequences of characters into *tokens* or *lexemes*. Section 2.1.1 introduced lexical analysis as carried out in most conventional compilers. The lexer for MSAD, is constructed from an LL(k) grammar specification, rather than a DFA. LL(k) grammars require that the input CFG be *left factored*, this procedure is described in the following Section 3.1.3. The ANTLR tool accepts this grammar specification and generates the lexer code, in our case the code for the `MScanner` class. `MScanner` is derived from the default `CharScanner` class supplied with ANTLR, but is given its own token list and methods. The token list is a listing of all the tokens that are identified by the lexer, in our case MATLAB keywords, identifiers, numbers and comments. These tokens supplied by the lexer using the `nextToken ()` method and forms the interface to the parser.

Keywords in MATLAB such as `if`, `for`, `function`, etc. are stored as string literals and are each assigned a unique integer token. For convenience the tokens in ANTLR based grammars are represented by a string of capital letters in the grammar specification. *White-spaces* and *new-lines* occur frequently in the source and are skipped for efficiency reasons in grammars for most languages. However, several MATLAB constructs such as invoking functions using the *command* syntax, or *matrix constructor* syntax are sensitive to both white-space and new-line characters. ANTLR provides a mechanism to filter the tokens through a token stream filter before the parser receives the tokens. In MSAD we filter the white-space (`WS`) and new-line (`NL`) tokens and add them to a hidden token stream which may be selectively read by the parser. This allows the parser to read the `WS` and `NL` tokens

only in the context of parsing constructs sensitive to these tokens. The other parser rules can simply ignore the hidden tokens. Other MATLAB operators, parentheses, curly and square braces, etc. are also assigned a token each by appropriate grammar rules. Rules to match numbers, identifiers, comments and strings are more complex and are built hierarchically by making use of regular definitions.

For example, simple identifiers can be matched using the simplified regular definition in Figure 3.1. This rule to match an identifier and generate the token `IDENT`, is based on two simpler rules `D` and `L` that match digits and letters respectively. MATLAB also allows an `'_'` in symbol names. The rule specifies a symbol (function or variable name) may naively be thought to be an alphabet followed by any number of alphabets, digits or `'_'`[1]. This rule has several limitations when it is used by the parser. The rule that matches an identifier does not test if the matched pattern is a keyword, in which case it should be assigned the corresponding token. This rule also does not distinguish between a variable and a MATLAB command name. Also, an `end` used in the context of an array indexing operation should be differentiated from an `end` used to terminate a `for` loop or an `if-else-elseif` construct.

```
   D  -> '0' | '1' | ... | '9'
   L  -> 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'
IDENT -> L ( L | D |'_' )*
```

Figure 3.1: Regular definition for an *identifier*

The complete rule to identify MATLAB identifiers in MSAD and disambiguate between ordinary identifiers and commands, and between use of an `end` keyword in an array indexing operation from a use to terminate a `for` loop or an `if-else-elseif` body, is shown in Figure 3.2. Rather than decode the rule in its entirety, we will only look at the relevant parts of the rule. In an ANTLR scanner grammar tokens are marked in all capital letters, individual character sequences are enclosed in quotes, a production is usually enclosed in round brackets, and each production or a set of productions can

[1]In practice MATLAB limits the length of variable or function name to 63 characters or the value of the builtin `namelengthmax` for portability.

```
1     WS : ( ' ' | '\t' )+
2  IDENT : L ( L | D |'_' )*
3          ( { follow_command }?
4            (        /* early exit for IDENT if no WS follows */
5                (  { LA (1) != ' ' && LA (1) != '\t' }?
6                    /* a WS followed by an operator is not a command */
7                |  ( WS ( PLUS | MINUS | NOT | TIMES | MTIMES
8                         | RDIVIDE | LDIVIDE | MRDIVIDE | MLDIVIDE
9                         | POWER | MPOWER | LE | LTmang | GE
10                         | GT | EQ | NE | ELM_AND | ELM_OR | AND
11                        | OR | COLON TRANSPOSE | QUESTION | '\''
12                        ) ) =>
13                    /* identifier followed by WS and a continutation
14                       sequence is not a command */
15               |   ( WS ( '.' '.' '.' ) ) =>
16                    /* OR we find a WS not followd by '(' or '='
17                       or a statement separator */
18               |  ( WS ~('(' | '=' | ';' | '\r' | '\n' ) ) =>
19                  {
20                    maybe_command = true;
21                  }
22               |  /* empty */  )
23           ) |  /* empty */  )
24         {
25             /* if structure, any field name is allowed */
26           if (!follow_field)
27           {
28                 /* check if the identifier forms a keyword */
29               _ttype = testLiteralsTable (_ttype);
30                 /* relax END check for indexed arrays */
31               if((brace_nest_depth > 0 || bracket_nest_depth > 0)
32                  && _ttype == END)
33                 $setType (ARR_END);
34                if (maybe_command && _ttype == IDENT)
35                 $setType (COMMAND_ID);
36           }
37             /* look for a DOT before another field */
38           follow_field = false;
39             /* a postfix operator may follow an identifier */
40           follow_postfix = true;
41             /* look for another valid command prefix */
42           follow_command = false;
43         }
```

Figure 3.2: Complete ANTLR *identifier* rule to disambiguate between MATLAB identifiers, commands, and `end` usages

be accompanied by a set of actions enclosed in curly braces. The actions are defined in terms of C++ statements. In the rules in Figure 3.2, we make use of ANTLR quirks called *syntactic* and *semantic predicates* [PLW+04] to help dis-ambiguate between conflicting matches. Syntactic predicates allow infinite lookahead to dis-ambiguate between conflicting productions with a common prefix. A semantic predicate allows selecting a production on the basis of any user defined condition. MSAD maintains a small scanning state to track the context in which characters occur. The state variables `follow_field`, `follow_postfix`, `follow_command`, `brace_nest_depth` and `bracket_nest_depth` indicate if the following identifier may be a structure field name, if a postfix operator may follow, if a command may follow, the nesting depth of current identifier in enclosing braces, or brackets respectively. For example, line `(5)` in Figure 3.2 uses a semantic predicate to test if a white-space does not follow an identifier, in which case the identifier does not represent a command. The production on line `(7)` is a syntactic predicate that tests if an operator follows a white-space after an identifier. The identifier does not represent a command in this case. Line `(18)` uses a syntactic predicate to test if an identifier followed by a white-space is not a part of an assignment statement or a statement on its own. If the identifier is not a keyword, it represents a command. Line `(32)` tests for nesting depth of the identifier in brackets or braces. If we match an `END` token inside braces or brackets, it is used inside an array indexing operation and we assign the new token `ARR_END` to indicate this is a MATLAB array `end` expression.

Comments in MATLAB v7.0 onwards come in two flavours, the standard single line comment prefixed with `'%'` and a multi-line comment enclosed in `'%{'` followed by a new-line and `'%}'`. The ambiguity in matching a multi-line comment is that we need to include all characters in the comment lexeme until we find the terminating `'%}'`. MSAD also uses the comment pattern `'%!'` to assign attributes to variables. We will see this syntax in the following subsection. This special comment or directive needs to be assigned a separate token `SCOMMENT` for the parser to differentiate directives from regular comments. Figure 3.3 shows the ANTLR rules to help MSAD dis-ambiguate and correctly match single and multi-line comments, and the

MSAD directives. The first alternative on line (4) matches the MSAD directive that is assigned a token SCOMMENT and passed to the parser. The second alternative on line (5) matches a multi-line comment, and the third on line (17) a single line comment. The second alternative is complicated by the fact that a missing '%}' can cause the remainder of the text in a file to be regarded as a comment. To help identify this condition we maintain a state variable state to track the sequence of characters in the input stream and locate a complete terminating sequence '%}'. In case we find a %{ without a following new-line, matched by production on line (14), this is simply a single line comment. The third alternative matches a single line comment that could be trivially empty, as matched by the production on line (17), or a single line of regular text, matched by line (18).

```
1       WS : ( ' ' | '\t' )+
2      NL1 : ( '\r' | "\r\n" )
3   NOT_NL : ~('\r'|'\n')
4  COMMENT : (  "%!" { $setType (SCOMMENT); }
5            |  "%{" (WS)?
6               ( ( NL1
7                   ( ~('\r'|'\n'|' '|'\t'|'%'|'}')    { state = 0; }
8                      |  '%' { state = (1 == state)? state+1 : 0; }
9                      |  '}' { state = (2 == state)? state+1 : 0; }
10                     |  (' '|'\t') /* include white-spaces in comment */
11                     |  { state < 3 }? NL1 { state = 1; }
12                  )*    /* multiline comment */
13                )
14              |  ~(' '|'\t'|'\r'|'\n') ( NOT_NL )*
15                /* multiline comment prefix but a single line comment */
16              )
17           |  '%' (  /* blank single line comment */
18                  |  ~('!'|'{'|'\r'|'\n') ( NOT_NL )*
19                     /* normal single line comment */
20                  )
21           )
```

Figure 3.3: Complete MSAD ANTLR rule to match single and multi-line *comments*

Consider the rules in Figure 3.4 to match constants in MATLAB. Since MATLAB regards an *ellipsis* to be a continuation symbol, to allow statements to extend over multiple lines, we need to distinguish between a real number

and the continuation symbol, that have a common single character prefix '.'. A structure field dereference operation also uses a '.' to access or update a field member. The lexer uses an additional lookahead character to resolves this issue. The first alternative for NUMBER on line (5) matches a structure field dereference and is assigned the token DOT. The second alternative on line (9) matches a continuation symbol, the third on line (11) a real number starting with a decimal point, and fourth on line (14) an integer or a real. The fourth alternative uses a syntactic predicate to test if a real number pattern prefix follows a natural number, in which case together they form a real number. Each of the numeric patterns is allowed to have an optional exponent field. Also, if these numbers are suffixed by an i or a j, the token is updated to indicate an imaginary number.

```
1        D : '0' | '1' | ... | '9'
2     IMAG : ('i'|'I'|'j'|'J')
3  NATURAL : ( D )+
4 EXPONENT : ('d'|'D'|'e'|'E') ('+'|'-')? NATURAL
5   NUMBER : (  '.' { $setType (DOT);
6                     follow_postfix = false;
7                     follow_field = true;
8                   }
9            |  '.' '.' '.' ( WS )* NL1
10              {  $setType (CONTINUATION);  }
11           |  '.' NATURAL (EXPONENT)?
12              (IMAG! { $setType (IMAG_NUMBER); })?
13              {  follow_postfix = true;  }
14           |  NATURAL ( ('.' D) => ('.' NATURAL) )? (EXPONENT)?
15              (IMAG! { $setType (IMAG_NUMBER); })?
16              {  follow_postfix = true;  }
17           )
18           {   /* look for another valid command prefix */
19             follow_command = false;
20           }
```

Figure 3.4: Complete MSAD ANTLR rules to match *constants* and *line continuation*

3.1.3 Parsing

In the last section we saw how the lexer `MScanner` in MSAD processes a sequence of characters and identifies tokens like `IDENT`, `DOT`, etc. The parser attempts to group together a sequence of tokens to match a higher level syntactic construct, according to the specified grammar. A high level construct may be an expression that adds two variables, invokes a function with required arguments, accesses array members etc. In MSAD the parser `MParser` is derived from `LLkParser`, the base ANTLR parser, and the implementation is generated by ANLTR using the supplied MATLAB grammar. The output of this phase is an abstract syntax tree with minimal number of annotations. Before we delve in to the details of the generated syntax tree, let us look at some sample constructs from the parser in detail.

Similar to the scanner grammar, in an ANTLR parser grammar, tokens are marked in all capital letters, a production is usually enclosed in round brackets, and each production or a set of productions can be accompanied by a set of actions enclosed in curly braces. The actions are defined in terms of C++ statements. Non-terminals can be assigned to a set of productions and are usually given descriptive names all in small letters. Additionally the parser grammar has directives such as `'!'`, `'^'`, `'< >'` to control the construction of the output syntax tree.

```
1   identifier [bool is_lval] :
2         id:IDENT <AST=MASTSymbol>
3         {
4           if (is_lval)
5             #id->setType (LVAL_IDENT);
6         }
7       | VARARGIN <AST=MASTSymbol>
8       | VARARGOUT <AST=MASTSymbol>
```

Figure 3.5: MSAD ANTLR parser rule to match *identifiers*

Before looking at a more interesting case, let us look at a smaller supporting parser rule for an MSAD *identifier* as given in Figure 3.5. Here an identifier is either a token of type `IDENT`, `VARARGIN` or `VARARGOUT`. In case

of an `IDENT` the associated action on lines `(4-5)` test if the `IDENT` occurs on the left hand side of an expression and assign a different token `LVAL_IDENT` in this case. This differentiates between an `IDENT` on the RHS and the LHS. `VARARGOUT`, corresponding to the MATLAB `varargout`, is implicitly assumed to be on the LHS, and `VARARGIN` for `varargin` on the RHS. The additional directives `<AST=MASTSymbol>` control the class type of the AST node. We will ignore directives to control AST output at this point, and re-visit them separately.

Consider a more complex parser rule in Figure 3.6. This rule trivially matches an *identifier* on line `(2)` using the previous rule in Figure 3.5. More interestingly the rule also matches more complex MATLAB constructs such as *structure* dereferences, *array* accesses, *cell* accesses and *function* calls. The production on line `(3)` indicates an identifier can be followed by a `DOT` token, followed by an identifer. This matches the MATLAB structure deference syntax e.g. `my_struct.field_name`. Line `(8)` matches a dynamic structure field access e.g. `my_struct.(field_name_var)`. The associated actions on lines `(10-17)` determine how the output AST is generated, which we will look at later in this section. The production on line `(19)` matches a cell array indexing operation e.g. `cell_arr{index1, index2}`. Line `(30)` matches array accesses and function calls. In MATLAB the syntax for the two is the same e.g. `array_or_fun_name(arg1, arg2)`, where `array_or_fun_name` is an identifiers and `arg1` and `arg2` are expressions. In case of an array access, arguments `arg1` and `arg2` form indices into the array. We therefore can not differentiate between an array access operation and a function call using syntax alone. This issue will be dealt with in the following Section 3.1.4.

The MATLAB language grammar is not publicly available, and hence requires to be coded up from scratch. The nearest reference is the Octave [Eat11] grammar, that has a similar syntax to most MATLAB constructs. Since Octave uses Yacc supported LALR grammar and uses *left recursion* in its production rules, the grammar is not usable in MSAD's ANTLR based LL(k) parsers. MSAD refers to some of the common arithmetic expression parsing rules from the Octave grammar, which need to be converted to a suitable form before they can be of use. Left recursion is not

```
1   indx_struct_cell_mat_or_func :
2     identifier[is_lval]
3     ( dot:DOT^ <AST=MASTOprStructFieldIndx> identifier[is_lval]
4       {
5         if (is_lval)
6           #dot->setType (LVAL_DOT);
7       }
8     |!( DOT! L_RND_BRAC! field_expr:simple_expr R_RND_BRAC! )
9       {
10        if (is_lval)
11          #indx_struct_cell_mat_or_func =
12              #( #[LVAL_DYNAMIC_FIELD, ".()"],
13                 #indx_struct_cell_mat_or_func, #field_expr );
14        else
15          #indx_struct_cell_mat_or_func =
16              #( #[DYNAMIC_FIELD, ".()"],
17                 #indx_struct_cell_mat_or_func, #field_expr );
18      }
19    |!( L_CUR_BRAC! cell_indx:cell_indx_list R_CUR_BRAC! )
20      {
21        if (is_lval)
22          #indx_struct_cell_mat_or_func =
23            #( #[LVAL_CELL, "{}"], #indx_struct_cell_mat_or_func,
24               #cell_indx );
25        else
26          #indx_struct_cell_mat_or_func =
27            #( #[INDX_CELL, "{}"], #indx_struct_cell_mat_or_func,
28               #cell_indx );
29      }
30    |!( L_RND_BRAC! arr_indx:array_indx_list
31        /* Trap erroneous productions " (  ) {  }" or "(  ) (  )"
32            => cell or array indexing into an indexed array  */
33        { LA(2) != L_CUR_BRAC && LA(2) != L_RND_BRAC }? R_RND_BRAC!
34      )
35      {
36        if (is_lval)
37          #indx_struct_cell_mat_or_func =
38            #( #[LVAL_MAT, "()"], #indx_struct_cell_mat_or_func,
39               #arr_indx );
40        else
41          #indx_struct_cell_mat_or_func =
42            #( #[FUNC_MAT, "()"], #indx_struct_cell_mat_or_func,
43               #arr_indx );
44      }
45    )*
```

Figure 3.6: MSAD ANTLR parser rule to match *structure* dereferences, *array* accesses, *cell* accesses, *function* calls or *identifiers*

supported in ANTLR since it generates a top-down recursive parser §2.1.2. We therefore need to use a method to eliminate left recursion from the desired rules and convert it to an equivalent LL grammar. Also, since the ANTLR parser is predictive, we need to make use of *left factoring* [ASU86, p.178], and adjust the lookahead appropriately, to remove parsing ambiguities.

A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \Rightarrow A\,\alpha$ for some string α. Consider the production rules for `sum_expr` and `postfix_expr` in Figure 3.7, that have a left recursive definition. The non-terminal `sum_expr` has two productions on lines `(1)` and `(2)` that are of the form $A \rightarrow A\alpha$, as is the case with the non-terminal `postfix_expr` on lines `(5)` and `(6)`.

```
1       sum_expr  -> sum_expr PLUS sum_expr
2                  | sum_expr MINUS sum_expr
3                  | prod_expr
4
5    postfix_expr -> postfix_expr TRANSPOSE
6                  | postfix_expr CTRANPOSE
7                  | primary_expr
```

Figure 3.7: Productions with left recursion

Algorithm 5 from Aho, Sethi and Ullman [ASU86, p.177] removes left recursion from a grammar, and is used here to convert rules that use left recusion to their equivalent form suitable for use with ANTLR. The algorithm also uses a supporting method to eliminate immediate left recursion in a rule. The method starts by grouping productions of A where no β_i begins with A:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

A list of equivalent productions without the left recursion is then given by:

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A'$$

ALGORITHM 5: Eliminating left recursion in parsing grammar

Data: Grammar G with no cycles or ε-productions
Result: Equivalent grammar $G^{'}$ with no left recursion
begin
  Arrange non-terminal in some order A_1, A_2, $\cdots$, A_n
  for $j = 1$ *to* n **do**
    for $j = 1$ *to* $i - 1$ **do**
      replace productions $A_i \rightarrow A_j\gamma$ with $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \delta_k\gamma$,
      where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all productions of A_j
    end
    eliminate the immediate left recursion from A_i productions
  end
end

Applying Algorithm 5 to the productions in Figure 3.7 we get the equivalent set of productions in Figure 3.8. Note the left recursive productions have been removed. We can further simplify this grammar by right factoring

```
1          sum_expr -> prod_expr sum_expr_1
2        sum_expr_1 -> PLUS sum_expr sum_expr_1
3                    | MINUS sum_expr sum_expr_1
4                    |
5
6      postfix_expr -> primary_expr postfix_expr_1
7    postfix_expr_1 -> TRANSPOSE postfix_expr_1
8                    | CTRANSPOSE postfix_expr_1
9                    |
```

Figure 3.8: After eliminating left recursion from rules in Figure 3.7

and using the EBNF repetition extension `'*'`. For example, in Figure 3.8 if we factor the common suffix in productions on lines `(2),(3)`, and `(7),(8)` we can re-write `sum_expr_1` and `postfix_expr_1` as:

```
        sum_expr_1 -> ( ( PLUS | MINUS ) sum_expr )*
      postfix_expr_1 -> ( TRANSPOSE | CTRANSPOSE )*
```

The factored and simplified rules used in MSAD are given by Figure 3.9.

```
1       sum_expr  -> prod_expr ( ( PLUS | MINUS ) prod_expr ) )*
2
3    postfix_expr -> primary_expr ( TRANSPOSE | CTRANSPOSE )*
```

Figure 3.9: After factoring and simplifying rules in Figure 3.8

MSAD AST

The abstract syntax tree (AST) generated by an ANTLR parser is in the form of a *generalised linked list* [Knu97] with *down* and *next* references to the neighbouring nodes. All tree nodes in ANTLR are based on the generic C++ class `CommonAST` provided by ANTLR. The `CommonAST` class provides an interface to the token type and the lexeme generated from parsing the original source, together with the down and next links to immediate neighbours inherited from its base class `BaseAST`, that supports the building of the tree-data structure. The parser grammar provides extra directives to control how the output AST is built. ANTLR also provides a *tree parser* mechanism to traverse the generated AST. A tree parser is a grammar specification of the structure of the AST, much like a regular parser uses a grammar of the language to parse a source file. Rules in the tree parser match sections of the AST, and like the parser grammar, can be assigned a set of actions to perform on matching the required tree pattern in the AST.

MSAD uses its own class hierarchy on top of the ANTLR provided `CommonAST` to support both the AST and the IR. Although the AST is not the IR in MSAD, the AST and the IR are both based on tree nodes. The root of all classes in MSAD that are related to tree nodes is `MAST`. During the development of MSAD profiling of the parser code revealed a significant portion of tree parsing time was spent in traversing through the sibling lists. It was also not possible to traverse up the AST from a child node. The `MAST` class now maintains references to the parent and the previous sibling in the AST for more flexible and efficient traversal of trees. The MAST class also

maintains an additional reference to the last child of the current node, this proved to significantly cut down time to add a new child node or get to the last sibling of the current node.

Consider the MATLAB statement in Figure 3.10. The statement on line `(1)` represents invoking a function call, or evaluating an array index operation depending on the type of `foo`. The MSAD scanner-parser generates an AST as shown in Figure 3.11. Note that the RHS of the assignment in the

```
1       result = foo(arg1, arg2, arg3);
2
```

Figure 3.10: Sample statement

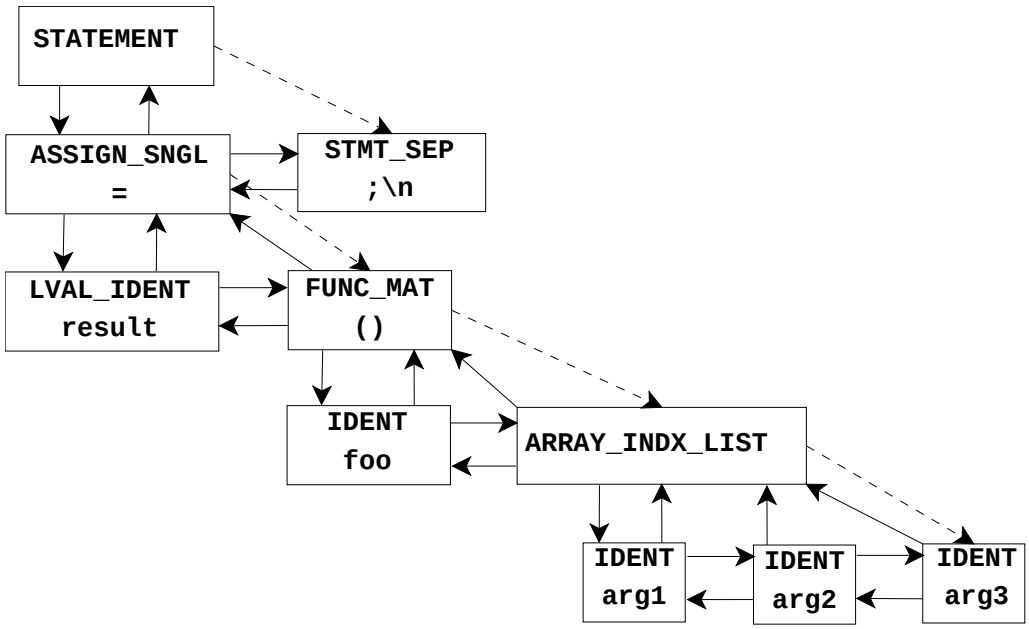

Figure 3.11: MSAD AST for statement in Figure 3.10

statement on line `(1)` in Figure 3.10, is matched by the parser production on line `(30)` in Figure 3.6. The `'!'` in the production on line `(30)` indicates to the parser generater to switch default AST construction off. The output AST is constructed from individual tokens in the actions associated with this

production on line (41). The root of the RHS sub-tree of the AST is set to FUNC_MAT as can be seen in Figure 3.11. The first child of FUNC_MAT is set to the current value of indx_struct_cell_mat_or_func, which is the leading token IDENT matched by line (2) of the parser rule. The second child is an argument list matched by the rule array_indx_list, not listed here for brevity. The AST shows all the links between neighbouring tree nodes (up, down, left, right, down_last_right), the token type and the associated lexeme for each node. In addition to making tree traversals more efficient, these extra references allow MSAD to implement its own custom tree traversals, e.g. to search for LHS or RHS identifiers in a statement. These custom tree traversals are useful in MSAD when traversing Use-Def chains in DCE optimisation, in global and local copy propagation and generating target (MATLAB) code.

ANTLR also allows the tree nodes forming the AST to be of different types, although sharing a common ancestor class. An AST with different node types is called a *heterogenous* AST. As mentioned in the previous paragraph, MSAD uses its own class hierarchy to implement the tree nodes. In Figure 3.11 for example, the root statement node, STATEMENT is of class MASTCFGStatement, the assignment node, ASSIGN_SNGL is of class MASTOprAssign, all identifier nodes IDENT are of class MASTSymbol. This can be seen in Figure 3.5 where the token IDENT is followed by the class specifier <AST=MASTSymbol>. In Figure 3.6 the token DOT on line (3) is of class MASTOptStructFiledIndx. The advantage of using a diverse set of classes (all derived from MAST) is for example, to be able to re-use common operations such as adding a child, or a sibling provided by the MAST class, and have special operations like extracting the RHS or LHS value of an assignment operation restricted to assignment nodes.

Apart from improving tree traversing, the MAST class helps decouple the syntax from the semantics. The class holds a reference to the symbol record MSymbol for the symbol represented by the current MAST node. The symbol record field is only used for tree nodes representing symbols. In the SSA form of the IR, all tree nodes referring to the same symbol share the same unique symbol record. This greatly simplifies data-flow analysis and optimis-

ing transformations. We will look at the use of `MSymbol` symbol records in the section on symbol disambiguation 3.1.4, SSA IR and optimisations.

M-file syntax

The input to MSAD is a MATLAB M-file containing the program of interest. For the purpose of AD the input file contains the primary function of the program that needs to be differentiated. The primary function may call other sibling, nested or global functions. MSAD requires the active inputs and outputs in any function, to be marked explicitly using compiler directives in the original source program. To allow for possible optimisations the `size`[2] of any input or output variable may be specified. If possible, the sizes of intermediates in the code-list are determined and can be used in operation re-ordering, and making use of efficient operators or constructs.

The code fragment in Figure 3.12 shows a sample function definition from the original file together with the directives indicating the input y and the output *dydt* to be active. The parameter N is inactive and is specified to be a scalar.

```
function dydt = foo(t, y, N)
%!  active(y), active(dydt), size(N) = [1, 1]
        .
        .
```

Figure 3.12: Sample function definition with source directives

Applying AD or optimising input programs that span multiple files is presently not supported by MSAD, although support for this is straightforward to implement in the current MSAD framework. The generic optimisation framework in MSAD supports most of the MATLAB 7.0 syntax and semantics. For instance, to test syntax coverage we supplied the MATLAB provided M-file for `ode15s` to MSAD, which is approx 1000 lines of MATLAB code with assorted complex MATLAB constructs. MSAD parsed,

[2]Following the MATLAB definition of size

optimised (optimised in compiler sense, not apply AD) and generated the correct `ode15s` output file, which was verified against the original by manual inspection, running through MLINT and comparing the ODE solution of Burgers' ODE in Chapter 4.

In terms of differentiability, the MATLAB coverage is presently limited to that necessary for tests presented later in Chapter 4. This is only from the time constraints of this research. MSAD however allows modular extension of both syntax coverage and adding new features to AD or other software components. Constructs like switch cases, cell arrays, structures, function handles, varargin and varargout are not fully supported for AD. The input source is assumed to be generally free of syntactic and semantic errors. At present there is little error checking in place within the parser, however programmatic asserts are in place in most of the software which check data-invariants at various stages of optimisation and transformation. Only a subset of the AD code for MATLAB builtins from the MAD package is included, enough to support the tests presented in Chapter 4.

3.1.4 Symbol disambiguation and scope resolution

As indicated in Chapter 2, Section 2.3 MATLAB is a dynamically typed language and it is not possible to statically determine a unique type for all the symbols in a program. However, it is possible to statically determine all possible types of a symbol. In MSAD we use an inclusive type lattice, described later in Section 3.3.2, which allows all potential types of a symbol to be propagated through the program. Generally speaking, there are two principle types of symbols in MATLAB, `Array` or `Function`. It may happen that in the input program a symbol is both an `Array` and a `Function` like in the example in Figure 3.13. The symbol `bar` is conditionally defined to be an array on line `(3)` and defined as a sibling function on line `(7)`. The definition of `bar` that reaches the use in statement `(5)` depends on the value of `x`. In recent versions of MATLAB v7 and later, such ambiguities are flagged as an error. MSAD therefore safely assumes such ambiguities are not present in the input source.

```
1     function z = foo(x, y)
2         if x > 1
3             bar = rand(3,4);
4         end
5         z = bar(y);
6
7     function c = bar(a)
8         c = a * 2;
9
```

Figure 3.13: Unresolvable type ambiguity

In the absence of static ambiguity described in Figure 3.13, it is possible to disambiguate between array and function symbols when syntax alone cannot distinguish between the two. A simple test if a symbol is variable, is to check if a definition (assignment to the variable) exists in the scope of the function in which the symbol occurs. However, MATLAB allows nested functions to share variables with the ancestors of a given nested function. This makes it difficult to simply test if a symbol has an assignment statement associated. Nested functions also have specific visibility rules with respect to its siblings and children. The first step in our analysis is to represent the relationships between the primary function in an M-file and its sub-functions, and potential nested functions. MSAD builds a function scope tree to associate sub-functions and nested functions. An example is presented in Figure 3.14 and the corresponding function tree in Figure 3.15. In our implementation this tree is stored as a generalised linked list with `next`, `previous`, `parent`, `child` references, much like the AST in Figure 3.11. For reference, the exact class structure can be found in the `MSymbFunction` class of Figure F.2.

Coming back to our problem of disambiguating symbols, in MATLAB, symbols are resolved in the following order:

1. Variable (local, parent, or global scope)

2. Sub-function or Nested-function

```
1     function A(x, y)
2
3         function B(x, y)
4
5             function C(x)
6             end
7
8         end
9
10        function D(x)
11
12            function E(x)
13            end
14
15        end
16
17        function G
18        end
19    end
20
21    function F(x)
22    end
```

Figure 3.14: Nested functions example

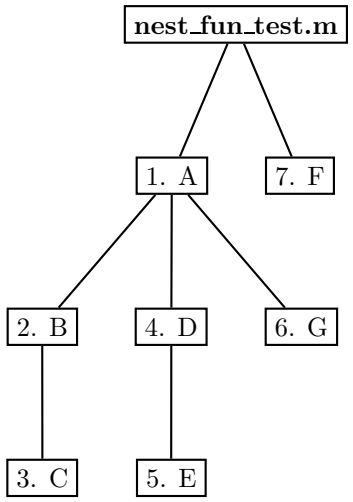

Figure 3.15: Nested function tree for example in Figure 3.14

3. Private function

4. Class constructor

5. Overloaded operation (resolved using the class of the arguments)

6. Function in the current directory

7. Function elsewhere in the environment path

As part of the process of disambiguating symbols, we also resolve the workspace of variables. MATLAB works with a notion of workspaces, which hold the associated data variables that scripts or functions operate with. In order to support scoping rules of sub-functions, nested functions and global variables, MSAD attempts to mirror MATLAB workspaces. Each function is allocated its local workspace and has access to the global workspace. Scripts by default work with the global workspace. The actual process of disambiguating symbols is performed in three steps given by Algorithms 6, 7 and 8 in order.

Algorithm 6 identifies all the symbols referenced in a function. The result of applying the algorithm to a function are five lists of symbols that are used (RHS) in the function, defined (LHS) in the function, form arguments to the function, form results of the function and global variables referenced in a function. This algorithm is applied to every function in a given file, the order here is irrelevant.

The resulting symbol lists are fed into the following Algorithm 7. This algorithm identifies variables and resolves the workspace in which they belong by traversing up the function tree looking for a definition of a symbol. A variable belongs to the workspace of the outermost function that defines it. The algorithm needs to be invoked on each function in the nested function tree in the in-order traversal order to operate correctly. The result is that the workspace of each function gets populated with variables that belong to that function. Global symbols are added to a special workspace called the global workspace, that every function or script share.

Finally, Algorithm 8 attempts to resolve any symbols that can not be disambiguated by syntax alone. The algorithm is only invoked on symbols

ALGORITHM 6: Mark symbol uses and defs

Data: Function f with list of statements S
Result: lists of symbols in function f – use_symbs, def_symbs, arg_symbs, res_symbs, and $global\_symbs$, that record uses, definitions, arguments, results and globals respectively

```
begin
    foreach symbol res_var in the result list of function f do
        res_symbs ← res_symbs ∪ {res_var}
    end
    foreach symbol arg_var in the argument list of function f do
        arg_symbs ← arg_symbs ∪ {arg_var}
    end
    foreach statement S_i in f do
        foreach symbol in S_i do
            if S_i is a globals list then
                globals_symbs ← globals_symbs ∪ symbol
            else if symbols is on the LHS (side-effect) then
                def_symbs ← def_symbs ∪ {symbol}
            else
                use_symbs ← use_symbs ∪ {symbol}
            end
        end
    end
end
```

that exhibit ambiguity. The result of the algorithm is the disambiguated class of the symbol, in this case `FUNCTION` or `ARRAY`. We will look at the MSAD class lattice in detail in Section 3.3.2. Adhering to the order of resolving symbols in MATLAB, the algorithm looks through workspaces of ancestors of the current function (in which the ambiguous symbol occurs). If found, the symbol must be a variable. Otherwise the algorithm searches the list of immediate children of the current function followed by the siblings of both the current function and its ancestors, and finally the list of global functions for a matching function corresponding to the symbol. If found the symbol must be a function. In the erroneous case where the symbol can not be determined to be either, it is assigned a generic `MCLASS` class. The caller routines handle this and generate an exception.

ALGORITHM 7: Resolve symbol workspace

Data: Function f_i in in-order traversal of the nested function tree, each with its associated symbols lists use_symbs, def_symbs, arg_symbs res_symbs, and $global\_symbs$; global workspace $root.global\_symbs$

Result: $workspace\_symbs$ list of symbols in workspace of function f_i

```
begin
    /* Resolve all symbols used or defined in this function */
    symb_list ← use_symbs ∪ def_symbs
    foreach symbol ∈ symb_list do
        resolved ← false, f_j ← f_i
        while resolved ≠ true do
            if symbol ∈ f_j.workspace_symbs then
                resolved ← true  /* Symbol found in workspace */
            else if symbol ∈ f_j.arg_symbs or symbol ∈ f_j.res_symbs
            then
                f_j.workspace_symbs ← f_j.workspace_symbs ∪ {symbol}
                resolved ← true  /* Symbol is argument or result */
            else if symbol ∈ f_j.global_symbs then
                if symbol ∉ root.global_symbs then
                    root.global_symbs ← root.global_symbs ∪ {symbol}
                end
                resolved ← true  /* Symbol is a global */
            else
                f_j ← f_j.parent  /* Move up the function tree */
            end
        end
        if resolved ≠ true and symbol ∈ f_i.def_symbs then
            /* This is the outermost definition of symbol */
            f_i.workspace_symbs ← f_i.workspace_symbs ∪ {symbol}
        end
    end
end
```

ALGORITHM 8: Disambiguate symbols (array or function)

Data: Symbol *symbol*, function f in which it occurs, workspace *workspace_symbs* associated with f, the global workspace *global_symbs*, and g the list of global functions

Result: *class* of *symbol* disambiguated between array and function

begin

```
/* 1.  Determine if symbol is a variable */
found ← false, f_i ← f, class ← "MCLASS"
if symbol ∈ globals_symbs then  class ← "ARRAY", found ← true
while found ≠ true and f_i is valid do
    if symbol ∈ f_i.workspace_symbs then
        class ← "ARRAY", found ← true
    end
    f_i ← f_i.parent
end
/* 2.  Determine if symbol is a function */
/*    a.  Test the immediate children of f */
f_i ← f.child
while found ≠ true and f_i is valid do
    if symbol = f_i then  class ← "FUNCTION", found ← true
    f_i ← f_i.next
end
/*    b.  Test the siblings of both f and its ancestors */
f_i ← f
while found ≠ true and f_i is valid do
    /* Get the first sibling in the sibling list of fi */
    f_j ← GetFirstSibling(f_i)
    while found ≠ true and f_j is valid do
        if symbol = f_j then  class ← "FUNCTION", found ← true
        f_j ← f_j.next
    end
    f_i ← f_i.parent
end
/*    c.  Search the list of global functions */
g_i ← g
while found ≠ true and g_i is valid do
    if symbol = g_i then  class ← "FUNCTION", found ← true
    g_i ← g_i.next
end
```

end

3.1.5 Medium-level intermediate representation

After the MSAD class lattice, which we will look at in detail in Section 3.3.2, the design of the medium-level intermediate representation (MIR) is among the most important aspects of MSAD. This is so primarily because all the analysis and transformations are based on the MIR. Where a good IR simplifies analyses and transformations, a badly designed IR will almost always make them difficult, even impossible. MSAD makes use of many generic SSA based optimisations in the internal framework all of which assume that the input is some form of an IR without complicated side-effects or semantics. In Section 2.2.1 we looked at the *three-address* form of the IR, where every complex expression was converted to a set of equivalent expressions, each of which performed exactly one operation on one or two inputs and generated a result. The examples in Figure 2.11 and Figure 2.13 used the MSAD IR to demonstrate how complicated expressions and control flow can be converted to simple three-address form. In this section we describe some of the complexities in converting MATLAB to an IR amenable to generic compiler analyses and transformations.

Lowering to MIR using tree re-writing

As mentioned earlier in Section 3.1.3, the AST is simply a mapping from the textual program input to a tree data-structure that is easier to process programmatically. The AST is not suitable as an IR because it preserves the complex syntax and semantics of the input high level language. MSAD transforms this parsed AST into a semantically equivalent tree or the MIR. MSAD uses ANTLR tree parsers in this lowering phase using an ANTLR feature called *tree re-writing* [PLW+04]. A tree parser as mentioned earlier in Section 3.1.3 is a grammar specification to traverse the AST in a programmed order. The tree parser processes the input AST and performs user defined actions associated with the grammar rules or even individual tree token nodes. In case of tree re-writing, the actions and directives control the mapping of input AST nodes to the output AST nodes. Tree nodes may simply be re-structured, their node token types re-newed (virtual tokens may be

assigned to increase the IR vocabulary if required), sections of the input AST may be deleted, or completely new sections addeded to the AST. Adding a new sub-tree to the AST is however limited to specifying the exact token sequence in a LISP like specification of the new tree to be added. Figure 3.16 is an example of a complete lowering rule in MSAD (simplified for clarity) for postfix expressions.

```
1  postfix_expr [bool fold=true] :
2    ( #(  TRANSPOSE
3          postfix_expr [fold]
4       )
5    | #(  CTRANSPOSE
6          postfix_expr [fold]
7       )
8    | (  indx_struct_cell_mat_or_func [fold]
9         {
10          /* Fixup pending command and function calls still masquerading
11           * as identifiers. If this identifier is a function i.e. it
12           * hasn't be lowered to FUNC_MAT during parsing, do so here.
13           */
14          RefMSymbol id_msymb;
15          if (#postfix_expr->getType () == IDENT
16              && (id_msymb = #postfix_expr->get_msymb ())
17              && id_msymb->get_mclass () <= "FUNCTION")
18            #postfix_expr = #( #[FUNC_MAT, "()"],
19                               #postfix_expr,
20                               #[ARRAY_INDX_LIST, "array_indx_list"] );
21          else
22            fold = false;
23        }
24     |  constructs [fold]
25        {  fold = false;  }
26     |  nest_expr [fold]
27        {  fold = false;  }
28     )
29   )
30   {
31     if (fold)
32       #postfix_expr = create_expression_temporary (#postfix_expr);
33   }
```

Figure 3.16: Tree parser rule to lower *postfix* expression in MSAD

Lines (18-20) in Figure 3.16 demonstrate one use of explicit tree construction with the ANTLR tree builder notation. The production on line

(8), indx_struct_cell_mat_or_func matches a structure, array or cell-array dereferencing operation, or a function call (this is assumed here for simplicity). The corresponding actions on lines (9-23) attempt to fix the IR for symbols that could not be disambiguated due to the common syntax of array variable access and function calls. These symbols were generically marked as variables using the identifier token IDENT during parsing. The symbol disambiguation phase, described earlier in Section 3.1.4, infers the class of such symbols' post-parsing. If a symbol was identified to be a function, the IR is changed at this point to match that of a function call i.e. a FUNC_MAT token parent node, and the function symbol and the list of arguments as children. The first node in the list is implicitly assumed to be the parent node. In the remainder of the code in Figure 3.16, the fold flag selectively enables the creation of expression temporaries to hold the result of the sub-expression, by invoking the action on line (32). The process of creating a temporary is delegated to the function create_expression_temporary. The rules constructs and nest_expr themselves store results in temporaries and therefore do not need the creation of another temporary here. The final result of the matching a postfix expression is therefore a single variable which may be assimilated by the parent in its own expression tuple. An expression tree is thus systematically broken down to a series of unary, binary or ternary operations.

Array `end` syntax

The MATLAB array end symbol can be used in an array indexing operations to represent the last index in the dimension in which it is used. MATLAB also allows indexing into a n dimensional array with an $n - m$ dimensional index, where the last dimension is implicitly expanded to include all the trailing dimensions. Semantically the value of an end is therefore dependent on three parameters: the associated array variable, the dimension in which it is used, and the total number of dimensions used in the indexing operation. The true number of dimensions is treated as an array property which is available through the array parameter. As the end symbol is only meaningful in the

context of an indexing operation, in the IR we convert this to a function call to `m_builtin_end` making the value independent of its syntactic context.

The lowering rule in Figure 3.17 shows how the three parameters are associated with the `end` to create a new function call to `m_builtin_end`. In many cases where new expressions or complete statements have to be added to the output IR, it is cumbersome to use the tree builder notations described earlier. In such cases MSAD re-uses its own scanner, parser, symbol resolution passes, and individual lowering rules (called recursively) to parse strings describing a statement or set of statements. This lowering example represents the smallest of the cases where MSAD re-uses its parsing infrastructure to generate IR statements during lowering. Lines `(13-14)` also show how a string representation of the replacement expression is built and passed into the `parse_string_statement` function which in turn invokes the scanner, parser, symbol resolution and lowering passes. The simplified expression tree is then extracted and set as the new output IR, lines `(17-20)`.

```
1   arr_end_expr [string array_var_name, uint arg_indx, uint nargs] :
2     arrEnd:ARR_END
3     {
4          /* ensure a valid array name has been assigned */
5        assert(array_var_name.length() != 0);
6          /* ensure the dimension index is valid */
7        assert (nargs >= 1 && nargs >= arg_indx);
8
9        RefMAST lower_end_mast;
10       stringstream lower_end_stream;
11         /* convert 'end' to
12          * 'm_builtin_end (array_var, end_arg_dim, num_args)' */
13       lower_end_stream << "m_builtin_end(" << array_var_name << ", "
14            << arg_indx << ", " << nargs << ");";
15       lower_end_mast = parse_string_statement (lower_end_stream);
16         /* get the expression from the statement */
17       lower_end_mast = lower_end_mast->get_first_child ();
18         /* remove the statement separator */
19       lower_end_mast->get_next_sibling ()->detach ();
20       #arr_end_expr = lower_end_mast;
21     }
```

Figure 3.17: Tree parser rule to lower array *end* expressions in MSAD

The IR in Figure 3.23, block `B5` shows how the `m_builtin_end` is used.

The complete operation on line `(5)` corresponds to the MATLAB syntax `index_vec(1:end,1:end)` or simply `index_vec(:,:)`.

Special variables `nargin` and `nargout`

MSAD also translates the MATLAB `narargin` and `nargout` syntax in to IR, similar to that of the array `end` operation described earlier, though the lowering logic for `nargin` and `nargout` is relatively straightforward. These variables hold at run-time the number of arguments passed in to a function and the number of results returned respectively. The `nargin` and `nargout` variables are, however, not explicitly defined in the input program, which confuses data-flow analyses. We therefore introduce explicit definitions of the two variables assigning them values returned by the synthetic functions `m_builtin_nargin` and `m_builtin_nargout` at the very start of the function (entry-block). This allows `nargin` and `nargout` to be treated as ordinary variables across the original user program. We will expand on the use of these `builtin` functions in Sections 3.3.2 and 3.4.

So far we have seen, using relatively simple examples, the precise method that MSAD uses to translate syntax from AST to MIR. These methods are generally applicable, and used throughout MSAD in increasingly complex scenarios. Rather than describing the complex implementation of lowering structures, cell arrays and loops, we will only discuss the resulting IR in the remainder of the section.

Structure field and cell array indexing

The lowering of high level data-structures like structures, cell arrays, and homogenous arrays, which can also be nested to arbitray depths, requires a more sophisticated scheme to map operation semantics. The IR needs to be designed taking into consideration the ease of analysis by making implicit dependencies visible. At the same time, the IR should be simple enough to manipulate without much overhead from the explicit dependencies. Consider the synthetic example in Figure 3.18 which demonstrates various operations on the structure `struct` supported by MATLAB. Line `(4)` shows a basic op-

```
1       function struct = test_struct()
2           struct = [];
3
4           struct.field1 = 1;
5
6           struct.details{1}(1:7) = 'string1';
7
8           struct.end = 'end';
9
10          temp = struct.details{1};
11
12          field = 'fieldname';
13          struct.(field) = temp;
14
15          field = 'details';
16          disp (struct.(field){1});
```

Figure 3.18: Synthetic test case to demonstrate lowering structure field dereferencing

eration of adding a field `field1` to structure `struct` and assigning a value of `1` to it. Line `(6)` shows a complex operation using nested structure, cell and array data-structures. The operation adds a new field `details` of type cell array to `struct`. The first element of the cell array `struct.details` is an array of characters that is assigned the value `'string1'`. Line `(8)` demonstrates the use of symbols that could otherwise be confused with the keyword `end`, but is simply a string literal giving the field name. MSAD handles field names specially in the scanner to avoid such ambiguity, see lines `(25-38)` of the scanner rule in Figure 3.2. In Figure 3.18, lines `(12-13)` demonstrate the use of dynamic fields where the field name is a variable rather than a literal string representing the field name. Finally lines `(10)` and `(15-16)` represent field and dynamic field dereferencing operations, however on the RHS of an assignment. Lowering all the operations in Figure 3.18 in MSAD results in the MIR in Figure 3.19. The numbers on the far right column in Figure 3.19 show line numbers of corresponding MATLAB statements in Figure 3.18.

Comparing the original program in Figure 3.18 with the IR in Figure 3.19,

```
1   function struct = test_struct()                                  (1)
2       struct = [ ];                                                (2)
3                                                                    (3)
4       struct = Struct_update(struct, 'field1', 1);                 (4)
5                                                                    (5)
6       tmp0 := Struct_access(struct, 'details');                    (6)
7       % implicit_update(struct);                                   (6)
8       tmp1 := Cell_ref_access(tmp0, 1, 'string1');                 (6)
9       tmp2 = 1:7;                                                  (6)
10      tmp1 = Array_ref_update(tmp1, tmp2, 'string1');              (6)
11      % implicit_use(struct);                                      (6)
12      % implicit_update(struct);                                   (6)
13                                                                   (7)
14      struct = Struct_update(struct, 'end', 'end');                (8)
15                                                                   (9)
16      tmp3 = Struct_access(struct, 'details');                     (10)
17      temp = Cell_access(tmp3, 1);                                 (10)
18                                                                   (11)
19      field = 'fieldname';                                         (12)
20      struct = Struct_dynamic_update(struct, field, temp);(13)
21                                                                   (14)
22      field = 'details';                                           (15)
23      tmp4 = Struct_dynamic_access(struct, field);                 (16)
24      tmp5 = Cell_access(tmp4, 1);                                 (16)
25      disp(tmp5);                                                  (16)
```

Figure 3.19: MIR for structure field dereferencing in Figure 3.18

we observe that all the operations have been made explicit using more descriptive tokens such as `Struct_update`, `Struct_dynamic_access`, etc. Every statement has a single operation and a fixed number of operands, much like the IR seen earlier in Figure 2.13 involving only arithmetic operations. The `Struct_update` operation on line `(4)`, for example, has three input operands, the original structure as the first argument, the field name to be updated as the second argument, and the value of the field as the third. The value of the field can be a constant (string or number) or a variable. The result of the update operation is a structure with the update operation applied. All inputs are treated as read-only, unless the input and output are the same.

On line (14) the field name `end` is seen to be correctly parsed by MSAD and identified as a field name to structure struct, with the value `'end'`. Line (20) shows the update operation in the original program using a dynamic field name is lowered to a `Struct_dynamic_update` operation. All operand types here are similar to the `Struct_update` operation earlier, except the second operand representing the field is simply a variable in place of a string literal. Lines (23-24) show the dynamic field access operation corresponding to the statement on line (16) in the original program. Note that all *access* operations have two input operands (variable and index), where the *update* operations have three (variable, index and value).

The two consecutive structure field and cell array access operations on lines (16-17) are semantically equivalent to the composite access operation on line (10) in the original program. The first `Struct_access` operation accesses the whole cell array, and the second `Cell_access` accesses the particular element of the cell array and copies the result in to the variable `temp`.

The composite dereferencing operation of nested structure, cell array and array in line (6) of the original program has been expanded to individual operations on lines (6-12) in the output IR. This operation is complicated by the fact that it updates a particular subset of the structure `struct` and yet, as a whole changes the value of `struct` completely. To simplify the nested update operations on the LHS, we introduce the concept of a *reference*. A reference is simply a placeholder for a partially dereferenced LHS operation like `struct.details` on line (6) of the original code, or the variable `tmp0` in the IR. This is analogous to the notion of a *pointer* in languages like C, C++ or Fortran. A reference is obtained by using a special assignment operation `:=`, again comparable to the `address` operator `&` in C and C++. References have special access and update operations associated with them e.g. `Cell_ref_access` and `Array_ref_access`. The last operation in the sequence on line (10) represents the actual array update operation that assigns the string `'string1'`. In addition to lowering the nested update operation using references, we need to make the side-effects obvious. For example, the last operation on line (10) in reality updates the whole structure `struct` as mentioned earlier. This is indicated by adding an *implicit_update* note like

on lines (7) and (12). Similarly an implicit input dependency is indicated by an *implicit_use* note like on line (11). The precise use of these notes will become clear when we impose the SSA property on the IR in Section 3.2.4.

For loop

We now look at converting high level MATLAB control flow structures like `for` and `while` loops, and conditionals like `if-elseif-else` to semantically equivalent low level intermediate representation. In Chapter 2 we introduced the concept of control flow analysis and explained the need for a low level IR. We also saw some examples of IR corresponding to loops and conditionals. In this section we discuss the semantic equivalence of the lowered control structures.

Consider the MATLAB `for` loop construct in the synthetic example in Figure 3.20 with the loop index iterating in the inverted interval $[10, 1]$ with a stride of -1. It should also be noted that the final value of the index i is output at the end of loop on line (6). MATLAB for loops vary from loops in languages like C with respect to the final value of the index.

```
1       function y = foo()
2           y = 1.618;
3           for i = 10:-1:1
4               y = y + (y / (pi * i));
5           end
6           disp(i);
```

Figure 3.20: Synthetic test case to demonstrate lowering a `for` loop

Figure 3.21 shows the IR constructed by MSAD corresponding to only the loop structure, statements on lines (3-5). The loop is replaced by a sequence of statements and two explicit *jump* (`goto`) statements, on lines (11) and (20). The jump on line (11) is a conditional jump, and is subtly different from the kind used to explain the IR earlier in Chapter 2. The examples used in Chapter 2 explicitly used two targets for clarity, one if the jump were taken, and the other if not. MSAD IR uses only a single explicit target in a

conditional jump. MSAD has a notion of a default exit edge of a basic block, i.e. the edge leading to the next basic block if there is no jump statement at the end of the current basic block, or the last statement is an unconditional jump. In case the last statement is a conditional jump, the default exit edge is to the basic block that would be taken if the jump predicate were not satisfied.

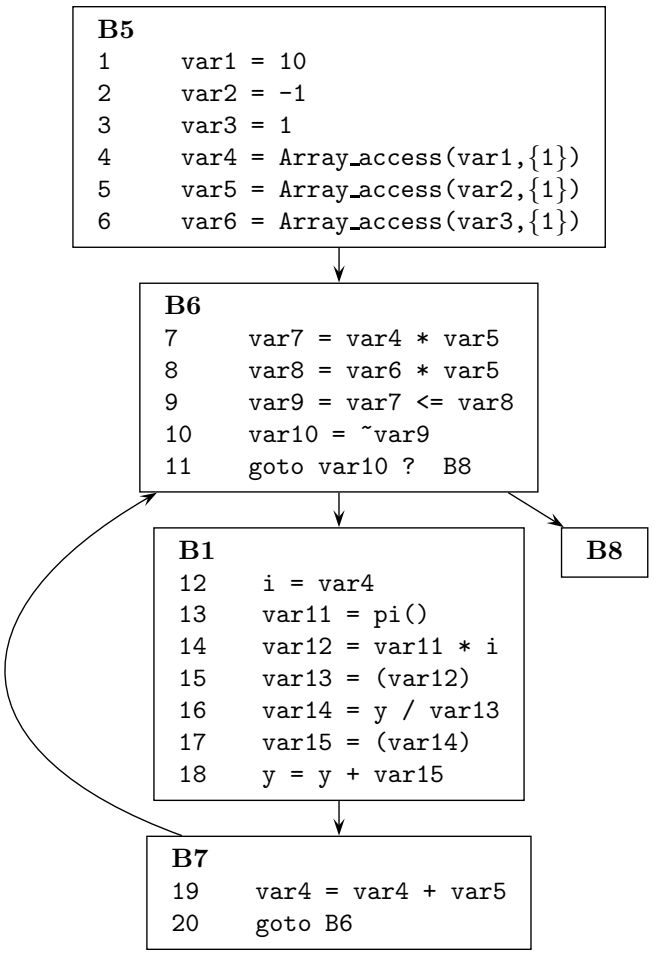

Figure 3.21: MIR for `for` loop in Figure 3.20

There are five basic blocks in the IR in Figure 3.21 generated from the original `for` loop structure in Figure 3.20. Block `B5` is called the *initialisation* block, `B6` is the *loop header*, *B8* is the *loop exit*, `B7` is the loop latch, and `B1` is the *loop body*. In many practical cases the loop body may span multiple basic

blocks. The initialisation block holds statements that are related to setting up the data-invariants at the start of the loop, such as the starting value of the loop index, `var1 = 10`, loop stride `var2 = -1` and the final value `var3 = 1`. In this example all three loop invariants are constants, but in the general case these may well be program variables. Because in MATLAB numeric variables can be arrays of any dimension, in the context of loop start, stride and limit, only the first element of the respective inputs are relevant. MSAD therefore scalarises these variables, statements `(4-6)`, before they are used.

The header block `B6` tests the loop continuation (or inversely the exit) predicate and conditionally branches to the exit block `B8`. By default the block that follows the loop header is the first block of the loop body, `B1`. The loop predicate should factor in the direction (sign) of the stride, so we can consistently test if the loop index is less than the loop limit value. We therefore multiply both sides of the inequality $index \leq limit$ by the *stride* value. This can be seen in statements `(7-9)`. Finally the predicate needs to be inverted to form a loop exit condition `var10` before the conditional jump in `(11)`.

The loop latch `B7` increments the loop loop index variable `var4` by the stride `var5` on each itertion of the loop and unconditionally jumps back to the loop header. This edge between the loop latch and the loop header is called a *back edge*.

At the very start of the first block, `B1` in the loop body, we copy the local loop index `var4` into the real loop index, `i` that may be used in the length of the loop body. We maintain this local loop index in addition to the original loop index in order to preserve the data invariants at the end of loop. In Figure 3.20 the value of i output at the end of the `for` loop is 1. Had we not maintained a separate local loop index, the final value would have been 0 because the loop index would be incremented, once additionally on the final iteration of the loop, by the stride in statement `(19)` in Figure 3.21.

We now look at another form of the `for` loop supported by MATLAB, that uses a single vector loop index. Figure 3.22 shows a similar synthetic `for` loop example as the earlier example. However in this case the loop index `i` is row vector of length 10 of random values in the closed interval $[1, 10]$.

The loop iterates over the length of the vector (specifically the number of elements in the column), in this case ten times. On each iteration the value of the loop index will be n^{th} element in the column (if the index is an array, the n^{th} column of the index array reshaped to 2D).

```
1       function y = foo()
2           y = 1.618;
2           indx_vec = floor(rand(1,10)*10) + 1;
3           for i = indx_vec
4               y = y + (y / (pi * i));
5           end
6           disp(i);
```

Figure 3.22: Synthetic test case to demonstrate lowering a `for` loop with a vector index

Figure 3.23 shows the IR corresponding to the `for` loop in Figure 3.22. Note that the overall structure of the loop is the same as in the earlier example in Figure 3.20. The only differences are in the handling of the loop index. In the initialisation block `B5`, statements on lines `(1)` and `(3)` compute the length of index array `index_vec` in their respective dimensions. The RHS of statement on line `(5)` is effectively `index_vec(:,:)`, and has the effect of reshaping the index array to two dimensions. The following paragraph explains how this reshaping is done. The LHS variable `var5` on line `(5)` forms the new correctly shaped index array variable. Variable `var6` computes the number of columns in the index array, or the *number of iterations* of the loop. The local loop index here is `var7` which is initialised to 1 and counts up to `var6` with a stride of 1. This monotonically increasing local index variable is used to index into the index array, as seen on line `(13)` inside the loop body, to extract the real index variable `i`. The loop predicate is also simplified by the strictly increasing loop index. The statement on line `(8)` simply tests if the local index variable `var7` is not greater than the number of iterations of the loop, `var6`.

The indexing operations in MATLAB treat trailing dimensions specially. For example, if `index_vec` were defined such that the call to the `rand` func-

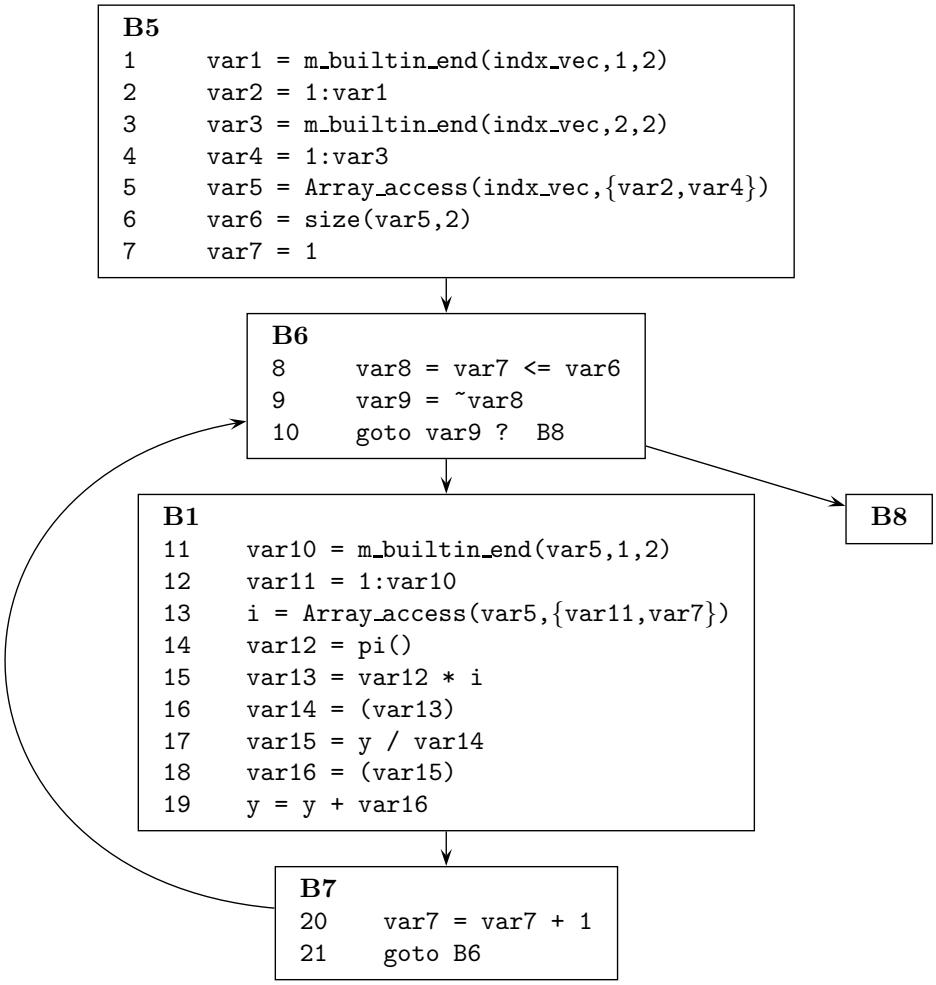

Figure 3.23: MIR for `for` loop in Figure 3.22

tion in Figure 3.22 was `rand(1,10,3)`, the value `var3` would be 30. This is the product of the length of all the trailing dimensions (two and three) including and after the dimension of the last index (two). In effect the operation on line `(5)` reshapes the index array to two dimensions, collapsing all the trailing dimensions after the second.

While loop

The program in Figure 3.24 is a synthetic test case to show the use of a MATLAB `while` loop and discuss MSAD IR code generation in this case.

The loop iterates while the predicate holds, in this case while the condition on line (4) is true. The initialisation of loop variable(s) such as the loop index is left to the programmer. In this example the relevant loop variable is `i`, which also the loop index in this case. The update of the loop variable(s) is also handled by the programmer. Interestingly the data-invariants at the end of the loop compared to a `for` loop, such as in Figure 3.20, are different. The value of i output at the end of the loop will be 11, a difference of one from the loop limit 10.

```
1       function y = foo()
2           y = 1.618;
3           i = 1;
4           while i <= 10
5               y = y + (y / (pi * i));
6               i = i + 1;
7           end
8           disp(i);
```

Figure 3.24: Synthetic test case to demonstrate lowering a `while` loop

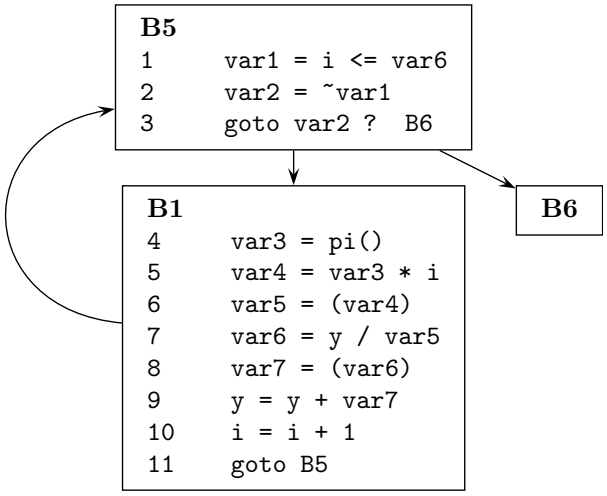

Figure 3.25: MIR for `while` loop in Figure 3.24

The IR for the `while` loop in Figure 3.24 is shown in Figure 3.25. We

note that the loop structure is considerably simpler than that for a `for` loop as seen in the earlier examples. The loop structure has three main blocks, the loop header `B5`, the loop latch `B1` and the exit block `B6`. In this case the loop body is the same as the loop latch. In more complex cases where the loop body contains other control flow, there may be several blocks in the loop body, the last of which has the unconditional jump to the loop header, or the back-edge, is the loop latch. The loop predicate is simply evaluated in the loop header block, statement on line `(1)` which is inverted on line `(2)` to form the conditional loop exit test. The data-invariant at the end of the loop is also preserved as the loop variables remain unchanged.

If-Elseif-Else conditions

The MATLAB `if` control construct allows conditional execution of nested statements, gated by a predicate. Paired with an `else`, the `if-else` construct allows exclusive execution of nested statements on either branch gated by a single predicate. This can be extended further by adding any number of intermediate `elseif` constructs each with their independent predicates to allows selective execution on a set of conditions. In the simple example in Figure 3.26, the input value `x` is tested for boundary conditions and *clipped* within the interval $[0, 255]$. This is done with two tests, one for each boundary, and a default case where the input is simply copied to the output. The order of testing the predicates and evaluating the nested statement, if the predicate is true, is strict.

The IR for the example in Figure 3.26 is given by Figure 3.27. The basic blocks `B0`, `B7` and `B2` hold the original statements nested in each of the `if`, `elseif` and `else` respectively. We also observe a unconditional jump `goto B5` added to block `B0` and `B1`. Block `B2` has a default exit to `B5`. Block `B5` is the common *exit* block for the `if-elseif-else` construct. Blocks `B6` and `B7` test the boundary condition predicates, corresponding to statements on line `(2)` and `(4)` in the original program in Figure 3.26. We also observe the order of testing the predicates and executing the associated statements, is preserved in the IR.

```
1       function y = foo(x)
2           if x > 255
3               y = 255;
4           elseif x < 0
5               y = 0;
6           else
7               y = x;
8           end
```

Figure 3.26: Test case to demonstrate lowering `if-elseif-else` conditional statements

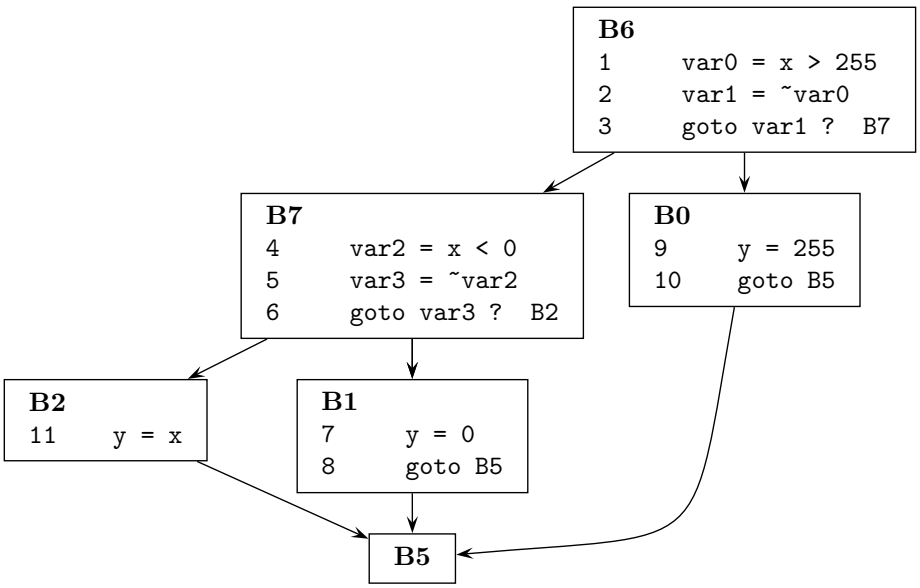

Figure 3.27: MIR for `if_elseif_else` conditional statements in Figure 3.26

Explicit loop control using `continue` and `break`

In case of loops, both `for` and `while`, MATLAB also allows control over individual iterations of the loop using explicit control constructs like `continue` and `break`. In most practical cases these constructs are conditionally executed and are nested within some form of an `if`, `elseif` or an `else`. Inside the loop body a `continue` statement causes the control to transfer to the start of the loop (skipping over the remainder of the loop body). In the case of a `for` loop the iterations continue normally, with the index variable being updated to the next value. In case of a `while` loop, the loop variable is maintained by the programmer and therefore an update to the loop variables if any, needs to be factored in by the programmer. A `break` statement on the other hand simply causes the loop to terminate.

MSAD lowers these `continue` and `break` constructs to IR in the form of explicit jumps (`goto`). In the case of a `continue` inside a `for` loop body, it is translated in to a jump to the *loop latch* which allows the index variable to be updated and the loop to resume naturally. In the context of a `while` loop the `continue` simply translates to a jump to the *loop header* because there is no need to manage the loop index update. A `break` statement inside a loop translates to a jump to the exit block, both in case of a `for` or a `while` loop.

3.2 Analysis

Once the input program has been converted from the high level syntax to lower level MIR as described in the earlier sections, we can apply many of the standard program analysis methods to gather data-flow and control-flow information about the input program. In this section we look closer at the implementation details of these analyses and the precise nature of this data- and control-flow information. Most of our work on MSAD focuses on the *intra-procedural* analysis and optimisations which forms the foundation of MATLAB program analysis and transformation framework. We do not claim to have implemented all the traditional compiler optimisations, in fact we have only selected the minimum possible to demonstrate the application

of AD. But the implementation allows a consistent method to introduce new analysis and optimisations. We describe our function specialising and inlining strategy as an *inter-procedural* optimisation, to apply AD to MATLAB programs. However, the extent of inter-procedural analysis is the bare minimum required to implement AD. The framework for this aspect is being extended.

3.2.1 Control flow graph

The basis of all intra-procedural analyses is the control-flow graph (CFG) upon which each analysis builds its own support data-structure. In Chapter 2 we introduced the concept of basic blocks, an algorithm to identify basic blocks and the control-flow graph with examples from the MSAD IR. The CFG in MSAD is straightforward with each basic block holding references to its successors and predecessors. Different analysis and optimisations may require different orders in which the CFG is traversed, such as depth-first, preorder, postorder or breadth-first. The successor and predecessor references help traverse the CFG in the desired order. Most analyses associate control- or data-flow information with basic blocks. In some cases, like Sparse conditional constant propagation optimisation described later in Section 3.3.2, we associate information with edges between two basic blocks. For this purpose MSAD selectively builds an edge-graph and allows a set of properties to be associated with the edges.

3.2.2 Live variable analysis

Live variable analysis, as described earlier in Chapter 2, is vital to the MSAD framework. It is used as a precursor in many transformations like converting MIR to SSA-MIR, discussed later in Section 3.2.4, coalescing SSA variable copies, and in converting MIR back to MATLAB code discussed in Section 3.5. Because the liveness information from this analysis is used at different stages of transformation, it needs to be re-computed as the transformations disrupt data-flow information. For this reason the algorithm should

be efficient. In this section we present the precise implementation of this analysis as used in MSAD.

Chapter 2, Section 2.3.3 explained the need for data-flow analysis, and described live variable analysis as a special case of backward data-flow analysis. The work list algorithm presented was a generic algorithm for forward flow problems. In this section we look at Algorithm 10 which shows the complete work list algorithm for backward data flow analysis with the flow function to compute liveness information in place. The liveness information is computed per block and propagated backwards starting from the `exit` block of a CFG. The inputs to Algorithm 10 are the CFG $G(V, E)$, and the pre-computed use, def bit lattices for all the blocks in V. We therefore start by looking at Algorithm 9 to compute the use and def lattices.

Muchnick [Muc97] defines $use(B_i)$ to be a lattice representation of the variables that are used in the basic block B_i before they are defined in the block, and $def(B_i)$ to be a lattice representation of the variables that are defined in the block before they are used in the block. Algorithm 9 processes each block B_i in the input program, and each statement in the block B_i in order. We use a temporary set M to track if a variable was encountered in an earlier statement in the same basic block, in which case no action is required. Unprocessed use-variables (on the RHS) are added to the use lattice and unprocessed def-variables (on the LHS) are added to the def lattice. In both cases the variable is also added to the temporary set M. At the end of the procedure we have lattices $use(B_i)$ and $def(B_i)$ as per their definitions.

Each block in the CFG is then processed by Algorithm 10 to compute the $lvOut(B_i)$ lattice property. This property represents the liveness of variables at the exit point of every basic block $B_i \in V$. The data-flow equations to compute the liveness property were discussed earlier in Section 2.3.3 (Equations 2.12 and 2.13). Equation 2.13 can be further substituted into Equation 2.12 to give:

$$lvOut(B) = \begin{cases} Init & \text{for } B = exit \\ \bigvee\limits_{S \in Succ(B)} (use(S) \vee (lvOut(S) - def(B))) & \text{otherwise} \end{cases}$$

ALGORITHM 9: Compute `use` and `def` bit lattices

Data: Program P with a set of basic blocks B
Result: $\forall B_i \in B$, $use(B_i)$ and $def(B_i)$ bit lattices indicating variables used and defined in B_i

```
begin
    foreach block B_i ∈ B do
        M ← φ        /* set of processed variables in B_i */
        foreach statement S ∈ B_i in order do
            foreach variable r used in statement S do
                if r ∉ M then
                    /* record use of r before any defs in B_i */
                    M ← M ∪ {r}
                    use(B_i) ← use(B_i) ∨ r
                end
            end
            foreach variable l defined in statement S do
                if l ∉ M then
                    /* record def of l before any uses in B_i */
                    M ← M ∪ {l}
                    def(B_i) ← def(B_i) ∨ l
                end
            end
        end
    end
end
```

In Algorithm 10 the combined equation translates to the operations on line 10 where the liveness, `use` and `def` information from all the successors of block v are combined to form the *lvOut* lattice value of the current block v. An important aspect of the work list algorithm is that the *lvOut* lattice of a predecessor only needs updating if any of the successors have changed. This condition is tested by the operation on line 12. More importantly the rate of convergence of this algorithm is highly dependent of the order of the basic blocks in the operation on line 3. Muchnick [Muc97, Ch.8] states that with the correct ordering of the basic blocks, the number of passes required for the algorithm to converge is bounded by $A + 2$, where A is the maximal number of back edges on any acyclic path in the flowgraph G. Almost always $A \leq 3$,

and mostly $A = 1$. For forward data-flow problems the optimal ordering of basic blocks is the *reverse postorder* of nodes in graph G. In case of backward data-flow problems the optimal ordering is simply *postorder*.

ALGORITHM 10: Live variable analysis using iterative worklist algorithm

Data: Flow graph $G(V, E)$ with set of nodes V, set of edges E, $exit \in V$, and sets $use(v)$ and $def(v)$ $\forall v \in V$
Result: $lvOut(v)$ $\forall v \in V$ where $lvOut(v)$ is the lattice of live variables at the exit of node v

```
1  begin
2      lvOut(exit) ← ⊥
3      Worklist ← N − {exit}
4      foreach v ∈ V do lvOut(v) ← ⊥
5      while Worklist ≠ φ do
6          b ← front(Worklist)
7          Worklist ← Worklist − {v}
8          T ← ⊥
9          foreach s ∈ Succ(v, G) do
10             T ← T ∨ use(s) ∨ (lvOut(s) − def(s))
11         end
12         if lvOut(b) ≠ T then
13             lvOut(v) ← T
14             Worklist ← Worklist ∪ Pred(v, G)
15         end
16     end
17 end
```

MSAD uses the following stack based recursive procedure AddReversePostOrder to generate the required ordering. To get the reverse postorder sequence of a CFG for example, the function `Next` is made to return the set of successors of input block, and the starting value v supplied to the procedure, is the `entry` block. In this case, the algorithm traverses the CFG with the `entry` block as root in a depth-first order adding a block to the stack once all its successors have been visited. The blocks are added to the stack in post order, but removing them from the stack reverses the order.

In the case of live variable analysis we wish to traverse the graph in

postorder sequence. We therefore use the same algorithm but invert the inputs. The function `Next` is made to return the set of predecessors of the input block, and the starting value v supplied to the procedure is the `exit` block. In this case the order of output is inverted, on account of starting from the `exit` and traversing backwards. The blocks are added to the stack in reverse post order sequence, but we access them in postorder sequence from the stack. On the CFG in Figure 3.27 for example the postorder sequence of blocks is: `B5`, `B5`, `B2`, `B1`, `B7`, `B0`, `B6`.

Procedure AddReversePostOrder(G, v, S)

```
Data: Flow graph G(V, E) with set of nodes V, set of edges E, v ∈ V
      the current node, and the node stack S
Result: Stack S with nodes in reverse postorder sequence
begin
    if HasTraversed(v) ≠ true then
        SetTraversed(v, true)
        foreach vn ∈ Next(v,G) do
            AddReversePostOrder(vn,G,v,S)
        end
        StackPush(S,v)
    end
end
```

3.2.3 Dominance analysis

In Chapter 2 we briefly explained the *dominance* relationship and its use in control flow analysis. In this section we look at the use of dominance information to construct *dominance frontiers*. In the next section we will see how liveness information and dominance frontiers help translate MIR to SSA-MIR by carefully placing the ϕ functions needed to merge SSA copies together. The SSA-MIR forms the platform for all transformations in MSAD. We start with a few more definitions for terms we will need to use in the remaining text.

Definition 3.1 (Dominance frontier) *The dominance frontier [CFR+91]*

DF(b) of a CFG node b is the set of all the nodes n such that b dominates a predecessor of n, but does not strictly dominate n.

DEFINITION 3.2 (STRICT DOMINANCE) *If node d of a flow graph dominates node n, and d ≠ n, then d is said to strictly dominate n, or d ≫ n. Plain dominance is denoted as d ≫̲ n. If d does not strictly dominate n, this is denoted by d ⋙̸ n.*

DEFINITION 3.3 (IMMEDIATE DOMINATORS) *The immediate dominator of a node n, written as idom(n), is the closest strict dominator of n on any path from entry to n.*

Algorithm 2 from Muchnick [Muc97, p.184] presented earlier in Chapter 2 provides a simple method to compute the dominators for a given CFG. However the time complexity of this algorithm is $O(n^2e)$ where n is the number of nodes in the graph and e is the number of edges. In large graphs the quadratic complexity term proves to be computationally expensive. The CFG of the non-vectorised form of Burger's ODE example discussed later in Chapter 4 has 27 basic blocks to begin. At the end of the AD augmentation process, the CFG has just over 3000 blocks. In order to efficiently update the dominance information, the dominance computations should be made more efficient. MSAD therefore implements a more complex algorithm by Lengauer and Tarjan [LEJ79] which has a complexity that is better than $O(e\,log_2 n)$.

The result of computing dominators is that every block in the CFG has an *immediate dominator* parent associated with it. All the blocks that are dominated by the same immediate dominator form the *dominator children* of that block. This parent-child relationship between the blocks (orthogonal to the CFG) forms a tree called the *dominator tree*. We saw one such example of a dominator tree in Chapter 2 Section 2.3.2.

We use Algorithm 11 by Cytron, Ferrante and Rosen [CFR+91] to compute the dominance frontier sets. The algorithm presented here also explicitly shows a stack based iterative method to traverse the dominator tree bottom-up as required by the original algorithm. The algorithm computes $DF(B_i)$

ALGORITHM 11: Compute the dominance frontier `DF(x)` of node `x`

Data: Flow graph $G(V, E)$ with set of nodes V, set of edges E, with $idom(v)$ and $domChildren(v) \forall v \in V$ pre-computed
Result: $DF(v)\ \forall v \in V$ where $DF(v)$ is the dominance frontier set for block v

```
1  begin
2      S ← φ      /* stack of pairs ⟨p,c⟩, p = idom(c), p,c ∈ V */
3      StackPush(S, ⟨v, FirstDomChild(v)⟩)
4      while IsEmpty(S) ≠ true do
5          pc ← StackPop(S), p ← pc.first, c ← pc.second
6          if c ≠ φ then            /* traverse down the dom tree */
7              pc ← ⟨p, NextDomChild(c)⟩
8              StackPush(S, pc)
9              StackPush(S, ⟨c, FirstDomChild(c)⟩)
10         else                  /* process block p on the way up */
11             DF(p) ← φ  /* initialise dominance frontier set */
12             foreach n ∈ Succ(p) do
13                     /* accumulate CFG successor effect */
14                 if idom(n) ≠ p then DF(p) ← DF(p) ∪ {n}
15             end
16             foreach m ∈ domChildren(p) do
17                 foreach n ∈ DF(m) do
18                     /* accumulate dominator children effect */
19                     if idom(n) ≠ p then DF(p) ← DF(p) ∪ {n}
20                 end
21             end
22         end
23     end
24 end
```

the dominance frontier set for a given block B_i in two steps. It accumulates contributions from the immediate successor blocks of B_i (line 14), and in the second step it accumulates the contributions from the dominator children of block B_i (line 19). This splitting of computation into two parts allows the computational complexity to be linear with respect to the accumulated size of the dominance frontier sets $\sum_n |DF(n)|$.

Figure 3.28 shows the result of applying the above algorithm to the CFG in Figure 2.21. The dominance frontier set for `B5` for example is `{B5,B4}`. The associated dominator tree is shown in Figure 2.22. Applying the definition of dominance frontier we can test that the dominance frontier sets are as expected. Consider the block `B6` for example, `B6` dominates itself which also happens to be the predecessor for `B4` but does not strictly dominate `B4` as there is another path from `B3` leading to `B4`. `B4` is therefore included in the dominance frontier set of `B6`. Another example is block `B5` which dominates `B6`, a predecessor of `B4`, but does not strictly dominate `B4`. `B4` is therefore a part of the dominance frontier set of `B5`. Another candidate is `B5` itself because `B5` dominates `B2`, a predecessor of `B5` but does not strictly dominate itself because of the path from `B0` to `B5`. Thus `B5` is a part of the dominance frontier set of `B5`.

3.2.4 Static single assignment based MIR

The MIR developed in Section 3.1.5 is useful as an IR because it is semantically equivalent, and yet simpler to analyse on account of each operation being isolated and the side-effects made explicit in individual IR statements. However, the MIR still lacks one important property we identified earlier in Chapter 2, Section 2.3.3, the absence of ambiguous definitions, that complicates data-flow analysis and hence the subsequent optimisations. MSAD therefore adopts the SSA form, which we introduced in Chapter 2, Section 2.3.3 that removes ambiguous definitions by providing a unique variable name to every assignment of the same variable in a procedure, and renames all the uses reached by an assignment to the corresponding new variable name. We also saw the concept of ϕ nodes that fuse together multiple SSA gener-

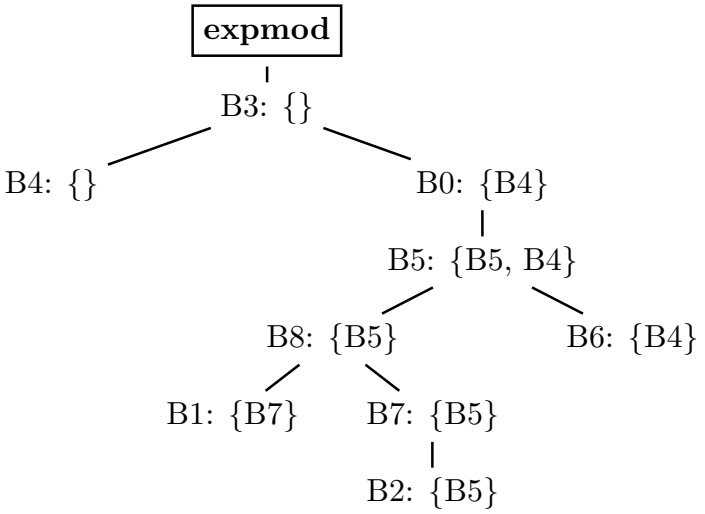

Figure 3.28: Dominance Frontier sets for CFG in Figure 2.21 and dominator tree in Figure 2.22

ated copies of same variable from different control-flow paths, to complete the data-flow graph.

The points at which the ϕ nodes need to be output have to be chosen carefully because a necessary, but missing ϕ node translates to wrong data dependencies, and hence the wrong program semantics. A surplus ϕ node although not catastrophic, hinders the flow of information during data-flow analysis and trivially increases the bulk of the IR. The problem of correct and optimum placement of ϕ nodes is therefore important. In this section we discuss our choice of algorithms to implement SSA in MSAD.

The method of translating to SSA as proposed by Cytron, et al. [CFR$^+$91] is carried out in three steps. The pre-cursor to the actual translation requires *dominance* information available in the form of a *dominator tree* and the *dominance frontiers*, which we have seen in the previous section. The next step in translating MIR to SSA-MIR is to place the ϕ nodes at the appropriate points in the CFG. The ϕ nodes in a basic block are all placed at the top, immediately after the *entry point* into the block. The last step in the translation is to rename every variable in the input program to include an SSA copy number which makes all variable definitions of the same variable

in the context of a function unique. The algorithm for renaming variables in MSAD, is the same as in Figure 12 of Cytron, et al. [CFR+91]. The complete algorithm to place ϕ nodes is given by Algorithm 12, which we will come to later.

```
1       function y = test_phi1(x, a)
2           y = 0;
3           if (x > 0)
4               t = x;
5               y = t .^ 2;
6           else
7               t = a;
8               y = t .^ 2;
9           end
```

Figure 3.29: Synthetic test to generate SSA variants

Here, we use a synthetic test case in Figure 3.29 to distinguish between two variants of the SSA form depending on the placement of ϕ nodes. The important observation is that there are two definitions of the variables `t` and `y` each along the two branches of the `if-else` construct on line `(3)`. The variable `y` is the result value of the function `test_phi`, and hence considered live in the `exit` block. The variable `t` however is not live outside the blocks constituting the two branches.

Figure 3.30 (a) shows the pruned SSA form of the code in Figure 3.29. Figure 3.30 (b) shows (the changed blocks from) the minimal SSA form. Comparing the two, we notice that block `B5` has an extra ϕ merging the copies `t_0` and `t_1` into `t_2`. However, the variable `t_2` is not used in the remainder of the code. Similarly we observe surplus ϕ nodes in the *exit* block `B4`, the results of which are not used. In this reduced example, the minimal form translates to three extra ϕ nodes. When parsing a large control-flow dominated code like the MATLAB `ode15s` function, for example, the minimal SSA form is approx. 2200 statements, out of which 821 are ϕ nodes. Whereas, in the pruned variant of the SSA only 444 statements are ϕ nodes. For this reason, MSAD uses the pruned variant of SSA by default.

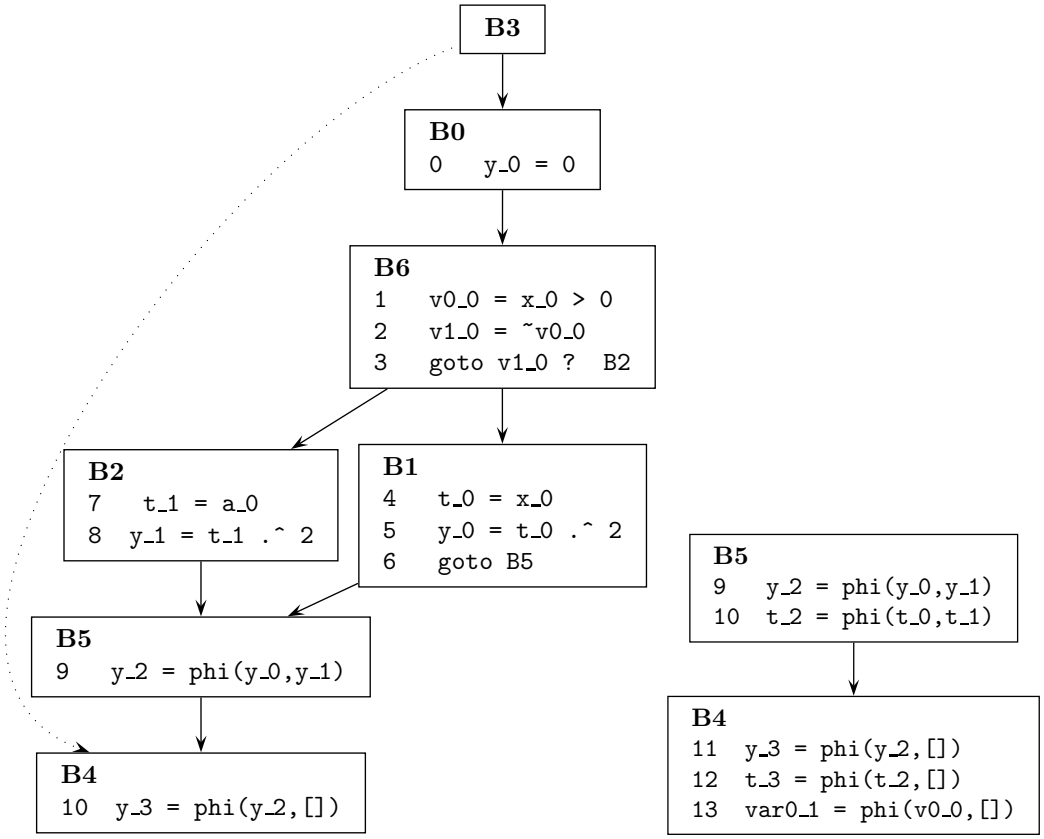

Figure 3.30: (a) pruned-SSA form, and (b) changes in the minimal-SSA form, for code in Figure 3.29

The original algorithm to place ϕ nodes, so as to construct the minimal SSA, is given by Cytron, et al. [CFR+89]. The modifications to construct the pruned SSA is given by Choi, et al. [CCF91]. Algorithm 12 shown here, lists the complete algorithm to place ϕ nodes and prune the ϕ nodes based on *live variable* information, as implemented in MSAD. This worklist algorithm iterates through the dominance frontier of each block that defines a variable (line 12), placing ϕ nodes in dominace frontier blocks (line 14) and their dominance frontier children (line 17). This forms the basic algorithm to construct the minimal SSA form. The operation on line 13 tests for liveness of the current variable in the dominance frontier block, before adding a ϕ node for it. This extra test essentially prunes unnecessary ϕ nodes. MSAD

provides a choice to build either, however it uses the pruned variant as default. Building the pruned SSA comes at an extra cost of having to gather *live variable* information but proves beneficial as it improves propagation of information in following optimisation passes.

ALGORITHM 12: Insert ϕ nodes

Data: Flow graph $G(V, E)$ with set of nodes V, set of edges E, $A(s)$, where $v \in A(s)$ if variable $s \in S$ is defined in node v, and $DF(v)$, $lvOut(v)$ $\forall\, v \in V$, where DF is a dominance frontier set, $lvOut$ is the live out lattice

Result: Appropriately placed ϕ statements fusing SSA variable copies

```
1  begin
2      /* NOTE: empty set is {} here, to avoid confusion with φ */
       count ← 0
3      foreach v ∈ V do
4          HasPhi(v) ← 0, Work(v) ← 0
5      end
6      W ← {}      /* W is a queue of blocks to be processed */
7      foreach s ∈ S do
8          count ← cout + 1
9          foreach v ∈ A(s) do  Work(v) ← count, W ← W ∪ {v}
10         while W ≠ {} do
11             v ← W.front()  /* pick v in front of the queue W */
12             foreach x ∈ DF(v) do
13                 if HasPhi(x) < count and s ∈ lvOut(x) then
14                     insert φ node, s ← φ(s, ··· , s) in x
15                     HasPhi(x) ← count
16                     if Work(x) < count then
17                         Work(x) ← count, W ← W ∪ {x}
18                     end
19                 end
20             end
21         end
22     end
23 end
```

Cytron et al. [CFR+91] also derive the computational complexity bound of the two algorithms used in this translation. Algorithm 12 has a complexity $O(A_{tot}\ avgDF)$ where $A_{tot} = A_{orig} + A_{\phi}$. A_{orig} represents the total number of

assignments to all variables on the LHS in the input MIR, and A_ϕ represents the number of new assignments generated by the ϕ statements in the program. The term $avgDF$ represents the average size of the dominance frontier sets computed in the previous section. The second renaming algorithm (not shown here) has a cost of $O(M_{tot})$ where $M_{tot} = M_{orig} + M_\phi$. M_{orig} represents the total number of uses of all variables in the input MIR, and M_ϕ the total number of uses in the newly added ϕ statements in the SSA-MIR. The complexity is therefore mostly linear with respect to program variables making this approach attractive as it simplifies many subsequent optimisations based on the SSA property.

3.2.5 Use-Def chains and SSA Defs

We introduced the concept of *ud-* and *du-chains* earlier in Chapter 2, Section 2.3.3. MSAD uses the SSA form of the MIR which has the useful property of making du-chains explicit, since every use is dominated by exactly one definition, the SSA-def. We therefore need to identify the unique definition corresponding to each use of a variable and associate it with that variable. The ud-chains on the other hand are useful to identify all the uses that are associated with a definition. The SSA-def and ud-chains are invaluable in most optimisations in MSAD. For example in MSAD, dead code elimination, sparse conditional constant propagation, copy and constant forward propagation, identifying induction variable and folding expressions when converting MIR to MATLAB code, all depend on ud-chains.

Our Algorithm 13 in MSAD used to identify the SSA-def and uses is relatively straightforward. We iterate through every statement and variable in a program once, associating the statement with the variable as a *SSA-def* if the statement defines this variable, or adding the statement to the variables *uses-set* if the statement uses this variable. The only two special cases at the start of the algorithm are that we explicitly initialise the uses of the result variables (line 3), and the defs of the argument variables (line 6) of the function being processed. To do this we have two placeholder statements $results\_use\_stmt$ and arg_def_stmt that explicitly use the result variables

ALGORITHM 13: Build Use-Def chains, and identify SSA-def

Data: Function f with list of statements S, placeholder statements $results\_use\_stmt$, and $args\_def\_stmt$
Result: Use set $v.uses$ and SSA-def $v.def$ updated $\forall v \in V$

```
1  begin
2      foreach variable 'v' in the result list of function 'f' do
3          v.uses ← v.uses ∪ {results_use_stmt}
4      end
5      foreach symbol 'v' in the argument list of function 'f' do
6          v.def ← args_def_stmt
7      end
8      foreach statement S_i in f do
9          foreach variable v in S_i do
10             if 'v' is on the LHS (side-effect) then
11                 if v.def = φ then v.def ← S_i   /* unique def S_i */
12                 else Error()      /* we expect a unique SSA def */
13             else
14                 v.uses ← v.uses ∪ {S_i} /* add S_i to set of uses */
15             end
16         end
17     end
18 end
```

and define the arguments respectively. We mark these statements immutable so they cannot be removed by any optimisation, but changed only by a few select optimisations like copy propagation.

Almost all transformations, as a part of optimisations, disrupt the ud-chain information and the SSA-def. The maintenance of this information is performed incrementally as transformation is applied. This saves the complexity of updating the complete ud-chain information after every transformation, or running the risk of using stale information.

3.2.6 Control Dependents

In Chapter 2, Section 2.3.2 we introduced the *dominance* and *postdominance* relationships, as defined by Definition 2.10 and Definition 2.11 respectively. In this section we introduce the concept of *control dependences* which is used

in the algorithm in the following section to implement *Dead code elimination* optimisation.

Definition 3.4 (Strict postdominance) *If node* `p` *of a control flow graph* ***postdominates*** *node* `n`*, and* `p` $\neq$ `n`*, then* `p` *is said to strictly postdominate* `n`*.*

Definition 3.5 (Immediate postdominators) *The* ***immediate postdominator*** *of a node* `n` *is the closest strict postdominator of* `n` *on any path from* `n` *to* `exit`*.*

Definition 3.6 (Control Dependent) *A node* `c` *is said to be* ***control dependent*** *[CFR+91] on node* `n` *in a CFG, if there is a non-null path* `p` *from* `n` *to* `c` *such that* `c` *postdominates every node after* `n` *on path* `p`*, and node* `c` *does not strictly postdominate node* `n`*.*

Cytron et al. [CFR+91] show how the concept of *control dependences* [FOW87] maps to the *dominance frontiers* of the CFG in reverse. To construct the dominance frontiers of the reversed CFG, we make use of Corollary 2.12 which allows us to re-use all the existing algorithms to construct the immediate postdominators, and the dominance frontiers of the reverse CFG. Instead of explicitly reversing the control flow graph we traverse the control flow graph in reverse. Because we store both predecessor and successor references for any block, it is trivial to traverse the CFG in reverse starting from the *exit* block in any required order. Algorithm 14 shows the steps involved in computing the postdominators *idom*, the postdominance frontiers RDF, and the control dependents CD.

Figure 3.31 shows the result of applying the above method to compute the postdominator tree and postdominance frontiers for the CFG in Figure 2.21. The main tree structure in Figure 3.31 is the postdominator tree, and the sets associated with individual blocks are the postdominance frontiers. In Figure 3.31 it is simple to verify block `B5` postdominates blocks `B2`, `B0`, `B7`, `B1` and `B8`. Any path from these blocks to the *exit* block `B4` must include block `B5`.

ALGORITHM 14: Compute control dependents, postdominators and postdominance frontiers of the CFG

Data: Control flow graph $G(V, E)$ with set of nodes V, and edges E
Result: $ridom(v)$ the immediate postdominator, $RDF(v)$ the dominance frontier in the reverse CFG, and $CD(v)$ the control depedents, where $v \in V$

```
1  begin
2  |  Apply algorithm by Lengauer and Tarjan [LEJ79] to compute the
   |  immediate dominators ridom of CFG G in reverse
3  |  Using ridom build the dominator tree of CFG G in reverse
4  |  Apply Algorithm 11 using ridom to compute the dominance
   |  frontiers RDF for CFG G in reverse
5  |        /* Now compute the control dependents CD(v) */
6  |  foreach v ∈ V do  CD(v) ← φ
7  |  foreach n ∈ V do
8  |  |  foreach v ∈ RDF(n) do
9  |  |  |  CD(v) ← CD(v) ∪ {n}
10 |  |  end
11 |  end
12 end
```

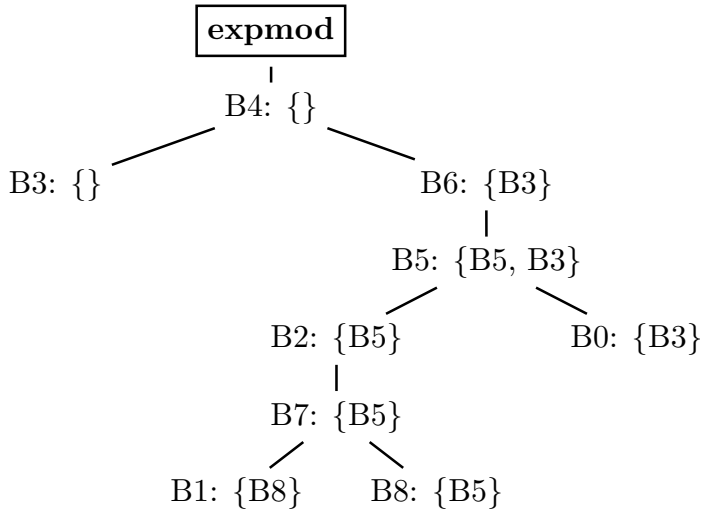

Figure 3.31: Dominance Frontier sets for **reversed** CFG in Figure 2.21

Table 3.1: Control Dependents using Algorithm 14 applied to CFG in Figure 2.21

b	CD(b)
B5	{B3, B7, B8, B2}
B6	{}
B7	{}
B8	{B1}
B0	{}
B1	{}
B2	{}
B3	{B5, B6, B8}
B4	{}

Table 3.1 lists the blocks in the CFG in Figure 2.21 and their control dependents, as computed by Algorithm 14. Consider block `B6` in the CFG in Figure 2.21 where `B6` postdominates blocks `B0` and `B5` but does not strictly postdominate `B3`, due to the edge from `B3` to `B4`. Block `B6` is therefore a control depedent of block `B3` according to Definition 3.6.

3.3 Optimisations

With the complete IR and necessary control- and data-flow analysis infrastructure in place, we now look into the optimisations MSAD applies to the IR in order to implement the specialising and inlining needed to apply AD. In this section we look at following main optimisations, dead code elimination, sparse conditional constant propagation (which includes class, sparsity, complexness, rank, dimensions and value inference), branch optimisations, constant folding, copy propagation and inlining.

3.3.1 Dead code elimination

We introduced dead code elimination earlier in Chapter 2. Here, with the necessary background to the analyses required prior to applying the optimisation, we can look at the algorithms used to identify and eliminate dead

code. MSAD uses three different methods to apply dead code elimination, which also includes unreachable code elimination. Unreachable code is removed very early in the optimisation passes, to prevent complicated and unnecessary processing of dead code. Dead code elimination is run twice in the normal optimisation-only phase, once immediately after the construction of the SSA form, and once towards the end of the optimisation pipeline as a part of copy propagation.

Unreachable code elimination

ALGORITHM 15: Unreachable code elimination

Data: Control flow graph $G(V, E)$ with nodes V and edges E, with entry block $entry \in V$
Result: Unreachable blocks $v \in V$ removed from the CFG

```
1  begin
2      repeat
3          changed ← false
4          foreach v ∈ V do
5              if Pred(v) = φ and v ≠ entry then
6                      /* Remove v from all its successors */
7                  foreach n ∈ Succ(v) do  Pred(n) ← Pred(n) − {v}
8                     /* Process every statement in this block */
9                  foreach statement s ∈ v do
10                     foreach variable x ∈ s do
11                             /* Remove use/def of x in s */
12                     if 'x' is on the LHS then  x.def ← φ
13                     else  x.uses ← x.uses − {s}
14                 end
15             end
16             Delete block v from G  /* Remove v from the CFG */
17             changed ← true
18         end
19     end
20 until changed ≠ true
21 end
```

Algorithm 15 removes any unreachable blocks from the input IR and is

run pre-SSA but after the construction of the CFG. The algorithm looks for blocks in the CFG that do not have any predecessors (line 5) i.e. they are not reachable in any manner. These blocks can be removed from the CFG. However, before a block v is removed from the CFG, all its succesor blocks n must be updated such that block v is removed from their predecessor set. Because the removal of a block may cause other successor blocks to be unreachable, we repeat the process until no blocks remain to be removed. Prior to removing a block we also process all the statements in the block, and all the variables in the statement to update the ud-chains and defs.

Control Dependents based Dead code elimination

Algorithm 16 by Cyrton et al. [CFR+91] is a SSA based variant of DCE used by MSAD to remove dead code. This `CD-DCE` algorithm uses SSA-defs, which we identified ealier, to propagate liveness of whole statements backwards through the data-flow graph. It also uses the control dependents information to propagate liveness information to ancestor blocks that the current block is control dependent on i.e. the inverse of control dependents or CD^{-1}. Cytron et al. [CFR+91] show that CD^{-1} is simply RDF or the dominance frontiers of the reverse CFG. We therefore do not compute the control dependents explicitly.

Algorithm 16 is optimistic in that it marks all statements dead to start with, except the obvious few. This *pre-live* set is determined according to the implementation. In MSAD we first mark the virtual statement that explicitly uses all the result variables, mentioned earlier in Section 3.2.5, in the exit block, live. This implies we wish to preserve all computations directly or indirectly linked to the result variables. We also mark MATLAB function calls or commands like `disp`, `error`, `warning`, `feval`, etc. live. Additionally, MSAD marks all branch statements as live.

In Algorithm 16, the $Definerss(s)$ function collects all the statements that define variables uses in statement s. This is done by simply traversing the statement looking for variables x on the RHS, and determining the statement that defines them, $x.def$. The function $Block(s)$ returns the block v

that the statement s belongs to, and function $Last(v)$ returns the last statement in block v. Finally, after deleting a dead statement s (line 27), all the variables used and defined in the statement should have their uses and def updated to remove s.

We present another synthetic test case, shown in Figure 3.32 to demonstrate the effect of applying Algorithm 16 to a MATLAB program. The function `test_dce` computes output `t` given inputs `x` and `n`. Clearly the computation of `op` inside the for loop on line `(12)` is redundant. This also implies the conditional initialisation of `op` on lines `(5)` and `(8)` is redundant, and so is the intermediate computation of `temp`.

```
1       function t = test_dce(x, n)
2           temp = [1:length(x)];
3           if any(x)
4               t = x .^ 2;
5               op = pi .* temp ./ 2;
6           else
7               t = zeros(n,1);
8               op = zeros(length(temp), 1);
9           end
10          for i = 1:n
11              t = t .^ (1/3);
12              op = [op; t];
13          end
```

Figure 3.32: Synthetic test case to demonstrate DCE

Applying Algorithm 16 to the input program in Figure 3.32 results in the optimised program in Figure 3.33. It should be noted, the `CD-DCE` optimisation is applied to the IR, but we have only shown the input and the final output here to simplify the example. Evaluating 1/3 to 0.33333333, on line `(8)` of the output, is done by MSAD's constant folding optimisation. We can also observe the remnants of SSA renaming (numbered suffixes) in the output code. However the SSA property clearly does not hold anymore, for example the definition of `t_0` on lines `(3)` and `(5)` share the same name.

ALGORITHM 16: Apply control dependents based dead code elimination

Data: Control flow graph $G(V, E)$ with nodes V and edges E, $RDF(v)$ control dependents, $PreLive$ set of statements always considered live

Result: Dead statements removed from blocks $v \in V$

```
1  begin
2      Live ← φ      /* mapping block → booleanflag, ⟨block, live⟩ */
3      foreach block v ∈ V do
4          foreach statement s ∈ v do
5              if s ∈ PreLive then Live(s) ← true
6              else Live(s) ← false
7          end
8      end
9      WorkList ← PreLive
10     while WorkList ≠ φ do
11         s ← WorkList.front
12         foreach d ∈ Definers(s) do
13             if Live(d) = false then
14                 Live(d) ← true
15                 WorkList ← WorkList ∪ {d}
16             end
17         end
18         foreach v ∈ RDF(Block(s)) do
19             if Live(Last(v)) = false then
20                 Live(Last(v)) ← true
21                 WorkList ← WorkList ∪ {Last(v)}
22             end
23         end
24     end
25     foreach block v ∈ V do
26         foreach statement s ∈ v do
27             if Live(s) = false then delete s from Block(s)
28         end
29     end
30 end
```

```
1       function t_0 = test_dce(x_0, n_0)
2           if any(x_0)
3               t_0 = x_0 .^ 2;
4           else
5               t_0 = zeros(n_0,1);
6           end
7           for i = 1:n_0
8               t_0 = t_0 .^ 0.33333333;
9           end
```

Figure 3.33: DCE applied to sample code in Figure 3.32

These copies have been reclaimed by a phase called copy coalescing, which we will see in Section 3.5.

3.3.2 Sparse conditional constant propagation

Constant propagation is a well-known and very useful compiler optimisation. The purpose of this optimisation is to discover constant values and propagate them to their uses with the intent that operations with constant operands will often simplify to a compile time constant value, or conditions gated by values which can be determined to be constants will be simplified. These simplifications save needless execution of code involving all constants at run-time, and often eliminates dead code. Many constant propagation algorithms such as simple constant propagation, sparse constant propagation, conditional constant propagation (see by comprehensive analysis by Wegman and Zadeck [WZ91]) have been used in the past. But, the *sparse conditional constant propagation* (SCCP) [WZ91] is most effective and efficient owing to the sparse data-flow graphs generated as a results of the SSA IR, and factoring in of conditional statements in the analysis which prevents (inferred) dead code from polluting data-flow information.

SCCP is one of the most complex optimisations applied in MSAD. The complexity arises not from the algorithm itself, but from the problem we attempt to solve using this algorithm in MSAD. Where most compilers e.g.

GCC [StG11], LLVM [LLV11] only use this algorithm conventionally i.e. to propagate constant values of variables, MSAD uses SCCP to propagate partial information on class, sparsity, complexness, rank, dimensions and value all at the same time. This requires a special value representation which we discuss later in this section. Propagating these attributes using the SCCP algorithms also needs lattice operations defined on all the attributes. In Chapter 2, Section 2.3.1 we introduced the lattice representations of all the attributes that MSAD carries out an inference for. The *class lattice* is more complicated and will be looked at separately in the following section. We use these lattice variables to propagate MATLAB variable attributes through the program. We start by looking at the worklists based algorithm first.

In all previous analysis we have assumed that the edges in the CFG are implicit from the predecessor-successor relationship. The SCCP algorithm associates an *executable* flag with each predecessor-successor pair. To be able to do this, we need edges to be explicit. We therefore build an edge-graph that records all such pairs prior to invoking the SCCP algorithm. In the context of the SCCP algorithm the CFG edges are called *flow edges*. The algorithm uses a second type of edge called an *SSA edge* to indicate data-dependencies that are effectively ud-chains.

Algorithm 17 is the main worklist algorithm, which is our implementation of the original SCCP algorithm by Wegman and Zadeck [WZ91], and uses three other support functions InitSCCP, VisitPhi and VisitExpression. Algorithm 17 also use two other functions VisitPhis and VisitBlock which simply iterate through the list of ϕs or other statements (exclusively) in a block, and invoke functions VisitPhi or VisitExpression on them. Although small, we have omitted the implementation for these, because the purpose of both the functions is obvious. Before we discuss Algorithm 17 we look at the support functions InitSCCP, VisitPhi and VisitExpression.

Function InitSCCP is merely meant to initialise the state of the variable used by the SCCP algorithm. As mentioned earlier the algorithm uses two worklists, the *flow work list* and the *SSA work list*. We initialise both to be empty and add all the edges leaving the `entry_block` to the flow work list. All the edges in the program are marked as non-executable to start with.

```
Function InitSCCP(SWL, FWL, P, G(V,E), entry_block, exit_block)
  begin
     /* NOTE: empty set is {} here, to avoid confusion with φ */
    FWL ← {}          /* initialise flow work list */
1   SWL ← {}          /* initialise SSA work list */
2   foreach se ∈ SuccEdges(entry_block) do
3     if se.dest ≠ exit_block then
4       /* add successor edges out of entry_block to the flow
           work list, except for the edge to exit_block    */
5       FWL ← FWL ∪ {se}
6     end
7   end
8   foreach e ∈ E do
9     e.executable ← false  /* set all edges non-executable */
10  end
11  foreach variable s in P do
12    /* initialise all variables to default class NULL     */
13    s.class ← NULL
14  end
15  VisitBlock(enrty_block)
  end
```

This function also initialises the default lattice state of all the variables in the given program P to be the MSAD class lattice type `NULL`. As we will see in the following section, MSAD uses an inclusive class lattice which implies the inference process determines all potential types of a variable, or the *lowest upper bound*, and not just a unique class, or the *greatest lower bound*. We therefore use the lattice join $\vee$ operation to combine attributes rather than the lattice meet $\wedge$. The correct starting value for the inference is therefore $\perp$ or `NULL` for the MSAD class lattice (not to be confused with `INVALID`). At the end of the initialisation the Function InitSCCP calls VisitBlock on the `entry_block`. This in turn has the effect of calling Function VisitExpression on the virtual statement, *arg_def_stmt* that defines all arguments to a function, which we came across earlier in Section 3.2.5. Processing this statement initialises all argument variables to be of type `ARRAY`, or in the context of AD, if any of the arguments are defined *active*, to be class `FMAD`.

ALGORITHM 17: Sparse conditional constant propagation

Data: Program P; Control flow graph $G(V, E)$ with nodes V and edges E, with *entry_block* $\in V$ and *exit_block* $\in V$

Result: Constants forward propagated to their uses

```
1  begin
2      /* NOTE: empty set is {} here, to avoid confusion with φ */
       InitSCCP(SWL, FWL, P, G(V,E), entry_block, exit_block)
3      while FWL ≠ {} or SWL ≠ {} do
4          if FWL ≠ {} then
5              fe ← FWL.front
6              if fe.executable ≠ true then
7                  fe.executable ← true, dest ← fe.dest, first ← true
8                  VisitPhis(dest, SWL)
9                  foreach pe ∈ PredEdges(dest) and first = true do
10                     if pe ≠ fe and pe.executable = true then
11                         first ← false   /* dest already processed */
12                     end
13                 end
14                 if first then VisitBlock(dest, FWL)
15                 if Length(SuccEdges(dest)) = 1 then
16                     FWL ← FWL ∪ SuccEdges(dest)
17                 end
18             end
19         end
20         if SWL ≠ {} then
21             se ← SWL.front, dest ← se.dest, blk ← Block(dest)
22             if dest is a φ then VisitPhi(dest, blk, SWL)
23             else
24                 executable ← false
25                 foreach se ∈ SuccEdges(blk) do
26                     executable ← executable ∨ se.executable
27                 end
28                 if executable then
29                     VisitExpression(dest, blk, FWL, SWL)
30                 end
31             end
32         end
33     end
34 end
```

Algorithm 17 iterates through both the flow work list and the SSA work list until either has an entry in them. The flow work lists are however processed differently to the SSA work lists. The general idea is the flow work list tracks updates to data variables that may change control flow, and the SSA work list tracks updates to all LHS variables that may change the result of all the statements that use them. The iteration continues until a steady state is reached where no updates occur to the lattice state of any variable, which implies the SSA work list becomes empty. Because no variables changed their lattice state, any control flow dependent on the variables will remain the same, hence the flow work list becomes empty. The convergence of this algorithm depends on the monotonicity of the lattice operations, which we discussed in Section 2.3.1.

Function VisitPhi shows how the SCCP algorithm combines the lattice state from all the SSA copies to infer the output lattice. Line 4 first computes the class type of the result of the ϕ. Assigning the class to a temporary lattice variable *op* initialises the internal value representation to create all other necessary attributes of the variable. Because all MATLAB variables (not symbols) are generically of type Array, we expect that the type of all variables will be at least the MSAD class latice type `ARRAY`. This implies each variable will have all the attributes of an Array such as sparsity, complexness, rank, dimensions and possibly a constant value. We initialise the lattice state of all the Array attributes to be initially *unknown* (`LOW_BOUND` in case of a boolean lattice, `NULL` in case of rank lattice and `Inv` in case of constants). We then compute the join of all the Array attributes to compute the final composite output lattice *op*. Line 24 tests if the output lattice *op* is different from that of the actual result variable of the ϕ statement. If the two differ, the lattice state of the ϕ result variable is updated to the new value *op*, and all the statements that use the result variable are added to the SSA flow list.

Function VisitExpression is similar to Function VisitPhi, but complicated by the fact that the semantics of combining operands in the expression depend on the operations in the IR vocabulary. MSAD uses an inference mechanism to compute the output lattice state of all the LHS variables in a statement, which is represented by the Function InferOutput. Because InferOutput deals

Function VisitPhi(phi_statement, block, SWL)

```
   begin
       opclass ← NULL
       argnum ← 0
       /* compute output class of φ as a join of input classes  */
1      foreach argument, arg in φ, phi_statement do
2          pe ← PredEdge(block, argnum)
3          if pe.executable = true and arg.class ≠ NULL then
4              opclass ← opclass ∨ arg.class
5          end
6      end
7      op.class ← opclass   /* ensure class is at least 'ARRAY' */
8      op.sparsity ← LOW_BOUND, op.complex ← LOW_BOUND
9      op.rank ← LOW_BOUND
10     op.size ← [Inv], op.value ← [Inv]
11     argnum ← 0
12     foreach argument 'arg' to φ 'phi_statement' do
13         pe ← PredEdge(block, argnum)
14         if pe.executable = true and arg.class ≠ NULL then
15             op.sparsity ← op.sparsity ∨ arg.sparsity
16             op.complex ← op.complex ∨ arg.complex
17             op.rank ← op.rank ∨ arg.rank
18             op.size ← op.size ∨ arg.size
19             op.value ← op.value ∨ arg.value
20         end
21         argnum ← argnum + 1
22     end
23     phires ← GetLHSLat(phi_statement)
24     if op ≠ phires then
25         /* set op to be the new φ output lattice              */
26         SetLHSLat(phi_statement, op)
27         foreach s ∈ GetLHS(phi_statement).uses do
           SWL ← SWL ∪ {s}
28     end
   end
```

Function VisitExpression(statement, block, FWL, SWL)

```
   begin
       /* InferOutput computes the output lattice of the LHS
          variables depending on the operation in 'statement', and
          returns 'true' if any output changed from its previous
          value, else 'false'.                                    */
1      changed ← InferOutput(statement)
2      if changed then
3          if 'statement' is a conditional branch then
4              target_edge ← GetBranchTargetEdge(statement)
5              if branch predicate in 'statement' is TRUE then
6                  FWL ← FWL ∪ {target_edge}
7                  foreach e ∈ SuccEdges(block) do
8                      if e ≠ target_edge then e.executable ← false
9                  end
10             else if branch predicate in 'statement' is FALSE then
11                 target_edge.executable ← false
12                 foreach e ∈ SuccEdges(block) do
13                     if e ≠ target_edge then FWL ← FWL ∪ {e}
14                 end
15             else
16                 foreach e ∈ SuccEdges(block) do
17                     FWL ← FWL ∪ {e}
18                 end
19             end
20         else if 'statement' is an assignment then
21             foreach variable 'lval' on the LHS in 'statement' do
22                 foreach use ∈ lval.uses do SWL ← SWL ∪ {use}
23             end
24         end
25     end
   end
```

with each operation in the MIR, we do not discuss this function in any more detail. The Function VisitPhi is a good representative of how attributes are combined together. It is worth pointing out that MSAD provides an extensible mechanism to provide inference of any number of MATLAB builtin functions such as `sin`, `reshape`, `det` etc. We have implemented a few minimum required for the purpose of specialising operations used in the MAD package. An interesting fact is a missing inference mechanism for a builtin merely implies MSAD uses a conservative analysis to compute the output lattice value, and not a failure of the inference mechanism.

Materialise SCCP constants

Once the SCCP algorithm has run, MSAD traverses through the SSA-MIR replacing any variables whose value is determined to be a scalar constant. This effectively materialises the forward propagated constant values at all the uses of the variable that is determined to be constant. Figure 3.34 shows a synthetic test case to demonstrate the application of SCCP in the presence of control flow. We observe that the value `t` is defined once outside conditional statements to a constant value `3 / 2`. The statement using `t` inside the `if` block computes the reciprocal of `t` and multiples by `pi` and assigns to `z`, all of which is effectively a constant. In the `else` block, `t` is re-defined to another constant matrix.

```
1       function z = foo(x)
2
3           y = ~isscalar(x)
4           t = 3 / 2;
5
6           if y <= 0
7               z = pi * (1 / t);
8           else
9               t = ones(2);
10              z = x * t;
11          end
```

Figure 3.34: Synthetic test case to demonstrate SCCP

The SSA-MIR for the program in Figure 3.34 is shown in Figure 3.35(a) and the output after the SCCP optimisation is shown in Figure 3.35(b). Looking at the output from SCCP we can see all the obvious constants have been folded and values of variables `t_0`, `MTmp3_0` and `z_2` have been forwarded and removed from the IR. The assignments to variables `MTmp2_0` and `MTmp4_0` are dead code, as there are no uses for these variables remaining. These statements are removed by DCE after SCCP. The final value of `z_2` has been forwarded into the ϕ. For the complete output of the SCCP iterations for this example, see Figure 115 which shows detailed output from the inference of each statement. Where no lattice updates are made, the inference information is simply not output.

Remove unreachable code

As we saw in the working of the SCCP algorithm and the following example, SCCP discovers and forward propagates constant values in the input SSA-MIR to Algorithm 17. Constants can be forwarded into regular assignment statements, into SSA ϕ nodes, or into conditional branch statements. If the predicate gating a conditional branch is discovered to be a constant, the branch can be statically determined to be taken or not taken depending on the value of the predicate. MSAD thefore runs through the SSA-MIR after SCCP and materialising the constants, and turns conditional branches into unconditional depending on the value of the predicate. This renders a portion of the CFG unreachable. MSAD uses a dominator tree based approach to efficiently remove the unreachable portion of the CFG.

If we modify the program in Figure 3.34 by adding the following MSAD directive `%! size(x) = [1 1]` to line `(2)`, MSAD will assume the input `x` to function `foo` will always be a scalar value. If we now look at the output MIR code after SCCP we can see most of the original code has been removed, and the only live code that remains is to compute the constant value of the output `z_0` on line `(23)`. On specifying the input to be a scalar, `MTmp0_0` in the MIR code of Figure 3.35(a) evaluated to 1, causing `y_0` to be 0 and hence the predicate on line `(12)` to be evaluated to be `true`. This implies

```
1  function z_0 = foo(x_0)
2    ENTRY_BLOCK_START 'MLbl3'
3    ENTRY_BLOCK_END 'MLbl3'
4
5    BLOCK_START 'MLbl0'
6      MTmp0_0 = isscalar(x_0);
7      y_0 = ~MTmp0_0
8      t_0 = 3 / 2;
9    BLOCK_END 'MLbl0'
10   % if
11     BLOCK_START 'MLbl6'
12       MVar0_0 = y_0 <= 0;
13       MTmp1_0 = ~MVar0_0;
14       goto MTmp1_0 ? MLbl2;
15     BLOCK_END 'MLbl6'
16
17     BLOCK_START 'MLbl1'
18       MTmp2_0 = pi();
19       MTmp3_0 = 1 / t_0;
20       MTmp4_0 = (MTmp3_0);
21       z_2 = MTmp2_0 * MTmp4_0;
22       goto MLbl5;
23     BLOCK_END 'MLbl1'
24   % else
25     BLOCK_START 'MLbl2'
26       t_1 = ones(2);
27       z_1 = x_0 * t_1;
28    BLOCK_END 'MLbl2'
29
30     BLOCK_START 'MLbl5'
31       z_3 = phi(z_2, z_1);
32     BLOCK_END 'MLbl5'
33   % end if
34
35   EXIT_BLOCK_START 'MLbl4'
36     z_0 = phi(z_3, null_ssa);
37   EXIT_BLOCK_END 'MLbl4'
```

(a)

```
function z_0 = foo(x_0)
  ENTRY_BLOCK_START 'MLbl3'
  ENTRY_BLOCK_END 'MLbl3'

  BLOCK_START 'MLbl0'
    MTmp0_0 = isscalar(x_0);
    y_0 = ~MTmp0_0
  BLOCK_END 'MLbl0'
  % if
    BLOCK_START 'MLbl6'
      MVar0_0 = y_0 <= 0;
      MTmp1_0 = ~MVar0_0;
      goto MTmp1_0 ? MLbl2;
    BLOCK_END 'MLbl6'

    BLOCK_START 'MLbl1'
      MTmp2_0 = pi();
      MTmp4_0 = (0.66666666666667);
      goto MLbl5;
    BLOCK_END 'MLbl1'
  % else
    BLOCK_START 'MLbl2'
      t_1 = ones(2);
      z_1 = x_0 * t_1;
    BLOCK_END 'MLbl2'

    BLOCK_START 'MLbl5'
      z_3 = phi(2.09439510239, z_1);
    BLOCK_END 'MLbl5'
  % end if

  EXIT_BLOCK_START 'MLbl4'
    z_0 = phi(z_3, null_ssa);
  EXIT_BLOCK_END 'MLbl4'
```

(b)

Figure 3.35: (a) SSA-MIR form for synthetic SCCP test case in Figure 3.34, (b) Output SSA-MIR after SCCP

the branch on line (14) is never taken. The SCCP algorithm by itself only substitutes a 0 in place of MTmp1_0 in the branch predicate.

MSAD uses the dominator tree to prune part of the CFG that is now dead from the conditional branch being changed to unconditional. In our example in Figure 3.35 the original conditional branch on line (14) of the input MIR was turned into a jump to the immediately following block MLbl6 which on its own is redundant and hence removed (because MLbl6 is the default exit of MLbl0). All the operations in MLbl6 were related to constants hence folded and removed. Using the dominator tree all the blocks rooted at block MLbl2 and below are redundant and hence removed. For the complete output of the

```
1  function z_0 = foo(x_0)
2    %! size(x_0) = [1, 1]
3
4    ENTRY_BLOCK_START 'MLbl3'
5    ENTRY_BLOCK_END 'MLbl3'
6
7    BLOCK_START 'MLbl0'
8    BLOCK_END 'MLbl0'
9    % if
10     BLOCK_START 'MLbl6'
11     BLOCK_END 'MLbl6'
12
13     BLOCK_START 'MLbl1'
14       z_2 = 3.141592653589793 * 0.666666666667;
15       goto MLbl5;
16     BLOCK_END 'MLbl1'
17   % else
18     BLOCK_START 'MLbl5'
19     BLOCK_END 'MLbl5'
20   % end if
21
22   EXIT_BLOCK_START 'MLbl4'
23     z_0 = phi(2.094395102393195, null_ssa);
24   EXIT_BLOCK_END 'MLbl4'
```

Figure 3.36: Output SSA-MIR after modifying program in Figure 3.34 by adding MSAD directive: `%!  size(x) = [1 1]`

SCCP iterations for this example, see Figure 94 which shows detailed output from the inference of each statement. Where no lattice updates are made, the inference information is simply not output.

Class Lattice

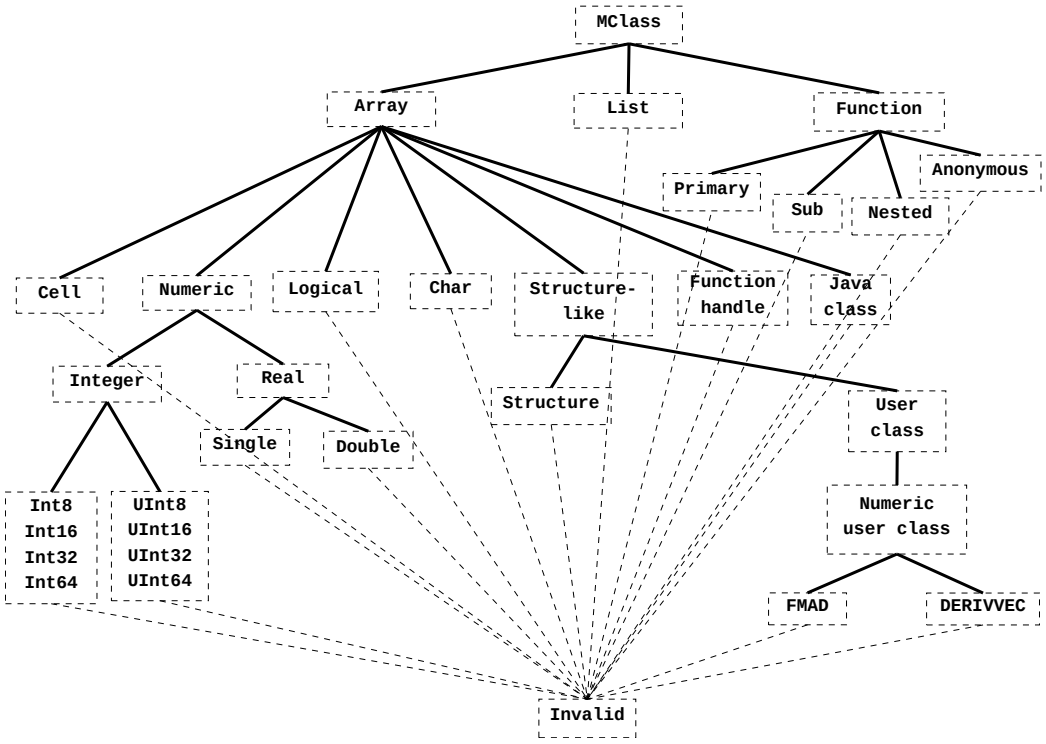

Figure 3.37: Simplified view of MSAD MATLAB class lattice

Figure 3.37 shows the leaf classes in the MSAD class lattice. This is a superset of the MATLAB Array classes [Mat11b] and contains classes like `MClass`, `Invalid`, `Function`, etc. which are not a part of the MATLAB class types. Like the lattice example shown in Chapter 2, Section 2.3.1, the classes in Figure 3.37 are only part of the set $\mathbf{S}$ that is used to determine the powerset $\mathcal{P}(\mathbf{S})$, which forms the complete MSAD lattice. The motivation to use the powerset $\mathcal{P}(\mathbf{S})$ instead of just $\mathbf{S}$ is to be able to track all the potential classes of a variable during program analysis. Each combination of the possible classes forms an element of the class lattice.

Figure 3.38 shows the implementation of the MSAD class lattice as a bit-lattice. Column 1 of the figure gives the index of the class which is used by Columns 4 and 5 to encode the child and sibling relationship (shown in Figure 3.37) between the leaves. Column 3 gives the number of leaves that

#	Bit-lattice	#Child	C	S	Class
0	00000000000000000000000000000000	0	0	0	NULL
1	00000000001111111111111111111111	22	2	30	MCLASS
2	00000000000001111111111111111111	19	3	26	ARRAY
3	00000000000000000000000000000001	1	0	4	CELL
4	00000000000000000000011111111110	10	5	17	NUMERIC
5	00000000000000000000000111111110	8	6	14	INTEGER
6	00000000000000000000000000000010	1	0	7	INT8
7	00000000000000000000000000000100	1	0	8	INT16
8	00000000000000000000000000001000	1	0	9	INT32
9	00000000000000000000000000010000	1	0	10	INT64
10	00000000000000000000000000100000	1	0	11	UINT8
11	00000000000000000000000001000000	1	0	12	UINT16
12	00000000000000000000000010000000	1	0	13	UINT32
13	00000000000000000000000100000000	1	0	0	UINT64
14	00000000000000000000011000000000	2	15	0	REAL
15	00000000000000000000001000000000	1	0	16	DOUBLE
16	00000000000000000000010000000000	1	0	0	SINGLE
17	00000000000000000000100000000000	1	0	18	FUNCTION_HANDLE
18	00000000000000000001000000000000	1	0	19	LOGICAL
19	00000000000000000010000000000000	1	0	20	CHAR
20	00000000000000001110000000000000	3	21	24	STRUCTURE_LIKE
21	00000000000000000100000000000000	1	0	22	STRUCTURE
22	00000000000000001100000000000000	2	31	0	USER_CLASS
23	00000000000000000000000000000000	0	0	0	NUM_CLASS
24	00000000000000100000000000000000	1	0	25	JAVA_CLASS
25	00000000000001000000000000000000	1	0	0	EMPTY
26	00000000000010000000000000000000	1	0	27	LIST
27	00000000001100000000000000000000	2	28	0	FUNCTION
28	00000000000100000000000000000000	1	0	29	FUNC_FILE
29	00000000001000000000000000000000	1	0	0	FUNC_INLINE
30	00000000010000000000000000000000	1	0	0	INVALID
31	00000000000000011000000000000000	2	32	0	NUM_CLASS
32	00000000000000001000000000000000	1	0	33	FMAD
33	00000000000000010000000000000000	1	0	0	DERIVVEC

Figure 3.38: The actual MSAD class bit-lattice

are children to any leaf class, and Column 6 gives the class names. The `ARRAY` class for instance has 19 leaf classes (does not include internal nodes like `NUMERIC` or `REAL`). All the necessary information is encoded into the bit-lattice in Column 2. The only purpose of using the remaining information is to initialise the class lattice at the start and add user defined classes, if any. The classes `FMAD` and `DERIVVEC` have been added to the default class lattice as seen in Figure 3.38. The duplicated `NUM_CLASS` entry on line `(31)` is the root of the user defined classes and over-rides the placeholder on line `(23)`.

In addition to the lattice join $\vee$, and meet $\wedge$ operations, the MSAD class lattice supports the standard partial ordering operations $<$ and $\leq$, and standard set predicates like *equals*, `intersects` and `subset` [RMG+00]. The MSAD class lattice provides a method `ub_mclass` to determine a leaf node that forms an least upper bound (LUB) on a lattice element (map LUB of $\mathcal{P}(\mathbf{S})$ to LUB of (S)). MSAD also provides a method `is_class` to determine if the MSAD class lattice may be of a specified type, this differs from simply using the lattice <= operation because the type to be tested and the type being tested may strictly intersect indicating an uncertainty.

The purpose of selecting an inclusive lattice can be demonstrated by applying some of the operations mentioned earlier to the class lattice. Figure 3.39 shows a typical usage of the class lattice within MSAD. All the variables prefixed *lattice_var* are MSAD class lattice variables. Assigning the type `DOUBLE` to *lattice_var*0 correctly initialises it to the bit-lattice value corresponding to type `DOUBLE` in Figure 3.38. On line `(4)` we join the type `LOGICAL` with *lattice_var*0 and confirm that the bit-lattice for *lattice_var*1 indicates both `LOGICAL` and `DOUBLE` types. Applying the `ub_mclass` operation to *lattice_var*1 correctly indicates the least upper bound type is `ARRAY`. We can confirm that `ARRAY` is truly the LUB by using the <= operation like on line `(8)`. We also show the use of `is_class` operation which indicates *lattice_var*1 is of `ARRAY`, not of type `CELL` and **possibly** of type `LOGICAL` on line `(11)`.

It should be noted that MSAD uses four distinct classes to indicate variants of unknown class. The MSAD classes `MCLASS` and `INVALID` represent the $\top$ and $\bot$ of a conventional lattice. A variable inferred to be of `MCLASS`

```
1     lattice_var0 <-- DOUBLE
2      => lattice_var0 =  00000000000000000000001000000000
3
4     lattice_var1 <-- lattice_var0 v LOGICAL
5      => lattice_var1 = 00000000000000000001001000000000
6
7     ub_mclass (lattice_var1) = ARRAY
8     (lattice_var1 <= ARRAY) = TRUE
9     is_class (lattice_var1, ARRAY) = TRUE
10    is_class (lattice_var1, CELL) = FALSE
11    is_class (lattice_var1, LOGICAL) = UP_BOUND
12    is_class (lattice_var1, FUNCTION) = FALSE
13
14    (REAL < NUMERIC) = TRUE
15
16    lattice_var2 <-- UINT64
17     => lattice_var2 = 00000000000000000000000100000000
18
19    ub_mclass(lattice_var2) = UINT64
20
21    lattice_var3 <-- lattice_var2 v REAL
22     => lattice_var = 00000000000000000000011100000000
23
24    ub_mclass (lattice_var3) = NUMERIC
```

Figure 3.39: Operations on the MSAD class Lattice

class would imply that the variable could be any one of the classes that are a children of `MCLASS`. A variable of class `INVALID` implies that the variable was inferred to be of two or more classes simultaneously, this is typically an error condition in the inference. Additionally MSAD uses a `NULL` class to serve as a starting point for the inference. A variable of class `NULL` has no class associated with it, and is treated as if in an intermediate state of class inference.

The physical realisation (software class in programmatic terms) of a MSAD symbol, once the class lattice value is determined, is shown in Appendix F in form of a class hierarchy of symbol type, storage type and the value representation.

3.3.3 Inlining

The previous section described how MSAD specialises MATLAB code by applying inference methods and forward propagating any inferred constants to their use. If enough partial information, about useful attributes like class, sparsity, rank, size and value, can be gleaned from a function it is often possible to optimised away large parts of the code that won't be executed. The previous example in Figure 3.36 demonstrated one such case where the information about the size (in MATLAB terms) of the input variable `x` was fixed and known $[1, 1]$. In that example the optimisation caused all of the function code to get optimised away, only preserving the constant value of the return variable. MSAD uses specialisation and inlining to implement AD precisely on this premise. We will see how the augmentation is done in the following section. Here we will look at the the method of inlining a callee into the caller.

In Chapter 2, Section 2.3.4 we gave an overview of the *inlining* optimisation with a simple example. We also discussed *late inlining* and some of the benefits of late inlining. We will discuss here how inlining is implemented in the context of SSA-MIR. To simplify this discussion we assume at this point that all functions that are called by the current function, have already been converted to the SSA-MIR form.

We discuss the inlining algorithm more abstractly, as a set of steps, to avoid confusion from detail:

1. Copy the lattice state of each *actual* argument (argument with which caller invokes callee at the call site) to the respective *formal* callee argument.

2. Set the constant value in the lattice state of *m_builtin_nargin* of the *callee* to the number of actual arguments.

3. Similarly set the constant value in the lattice state of *m_builtin_nargout* to the number of actual result variables at the call site.

4. Apply SCCP algorithm to the *callee*

5. Apply DCE to *callee*
6. Apply basic algebraic simplifications to *callee*
7. Insert copies from the the the actual arguments to the formal in the entry block of the callee, update uses of the actual arguments and SSA-defs of the formal
8. Insert copies from the formal result variables to the actual in the exit block of the callee, update SSA-defs of the actual results and the uses of the formal
9. Set the block type of the callee entry and exit blocks to be normal `CFG_BLOCK`s
10. Remove the virtual edge from the entry to the exit block
11. Split the current block (in the caller) by moving all the previous statements in this block to the head of the (now old) entry block of the callee
12. Delete the call statement from the current block (containing the function call)
13. Set all predecessors of this block to (now old) entry block of the callee
14. Set the successors of the (now old) exit block of the callee to the successors of this block
15. Update caller function data-structure tracking blocks and variables belonging to a function such as workspaces.

The important point to note here is that because we are inlining post-SSA, the SSA names needs to be unique across all functions to avoid clobbering unexpected variables due to overlap.

3.4 AD augmentation

The previous section 3.3 described several generic optimisations that MSAD applies to the SSA-MIR code. The augmentation pass described here harnesses all these generic optimisations to implement the forward mode of AD by specialising the `fmad` and `derivvec` class operations of the MAD package and inlining them in the input program.

As with any other source transformation AD tool, MSAD also needs programmatic hints to indicate which variables from the inputs to treat as independent. The output derivatives are computed with respect to these independent variables. The MSAD `active` directive is used as shown in the synthetic test case in Figure 3.40, and indicates that the input variable `x` is an independent variable, or within the MSAD framework, of type `FMAD`. The derivatives corresponding to these variables will be of type `DERIVVEC`. The type inference from SCCP determines the potential classes of all variables in the input program. In the context of AD the type inference also serves as activity analysis, as any variable that could potentially be of `FMAD` class should be treated as active. Additional information regarding the sparsity of the input derivatives is supplied using the `sparse_der` directive. MSAD's sparsity inference propagates this attribute through the data-flow graph. MSAD can also be directed using command line arguments to strictly choose *full* or *sparse* storage for derivatives.

From the inlining procedure outlined in the previous section we saw how MSAD transfers lattice properties from the call-site in the caller to the callee. In case of AD, the callee functions will be the `fmad` and `derivvec` operations. Once a variable is inferred to be potentially of class `FMAD`, MSAD specialises and inlines the relevant operation into the CFG of the caller function (or the function to be augmented in the case of AD). Once all the operations with variables of type `FMAD` class have been processed, the augmented SSA-MIR represents a differentiated form of the original SSA-MIR.

Although MSAD is able to process all of the MAD class operations as is, MSAD presently lacks the *Scalar replacement of Aggregates* (SRA) [Muc97, Ch.12] pass to be able to collapse the class structure and generate code

amenable to MATLAB JIT acceleration. This pass simply turns fields of a structure into separate variables. Due to time constraints of this research we decided to be pragmatic and work around this by generating a pre-processed MATLAB form of the MAD operations. An example can be see in 3.42. Once the SRA pass is in place, we will simply revert to using the original MAD code. Other than applying SRA, the MAD code is is no way simplified and preserves the full complexity of the original MATLAB code.

Appendix C.1 and Appendix C.2 show complete output from MSAD in generating first order derivatives for the MINPACK-2 enzyme reaction problem, and MATLAB Brown's minimisation problem respectively. The results from several other tests successfully differentiated and tested are discussed in Chapter 4.

```
1  function y = test_for(x, a)
2  %! active(x)
3
4    y = a;
5    for i = 1:10
6      y = y + x * (pi / i);
7    end
```

Figure 3.40: Synthetic test for testing AD augmentation

For the purpose of demonstrating the augmentation process we use a simple synthetic example shown in Figure 3.40 that uses a `for` loop, and two active arithmetic operations `plus` and `times`. The final output from MSAD after augmenting the SSA-MIR and generating MATLAB code can be seen in Figure 3.41. The complete intermediate SSA-MIR is very large to include here to explain the augmentation process. However, it is possible to look at a single operation in isolation. Figure 3.42 shows the `plus` operation from MAD after applying SRA as explained earlier. MSAD infers the `plus` operation on line `(6)` in Figure 3.40 to be active, because both operands `y` and `x * (pi / i)` are determined to be of class `FMAD`. The output code in Figure 3.41 lines `(27-35)`, and the SCCP specialised SSA-MIR of the plus operation shown in Appendix D.2 confirm this. The specialised SSA-MIR also shows the code in the original `plus` operation in Figure 3.42 lines `(26-37)`, which handles

the cases where strictly either input operand is active, has been optimised out. The predicate testing the class of the inputs on line (5) in Figure 3.42 is also statically inferred to be `true` and removed from the SCCP output.

```
1   function [y_0, ad_y_0] = ad_test_for(x_0, ad_x_0, y_0)
2
3       n_derivs_2 = floor(numel(ad_x_0) ./ numel(x_0));
4       ad_x_1 = reshape(ad_x_0, [numel(x_0),n_derivs_2]);
5
6       MVar5_0 = floor(numel(ad_x_1) ./ numel(x_0));
7       ad_y_0 = zeros(numel(y_0), MVar5_0);
8       for MVar4_1 = 1:10
9           MTmp5_0 = (3.141592653589793 / MVar4_1);
10          z_1 = x_0 .* MTmp5_0;
11
12          n_derivs_1 = floor(numel(ad_x_1) ./ numel(x_0));
13
14          tmp_ssb__1 = numel(x_0);
15          if 1 == tmp_ssb__1
16              d_z_1 = MTmp5_0(1, ones(1,n_derivs_1)) .* ad_x_1;
17          elseif tmp_ssb__1 == 1
18              d_z_1 = MTmp5_0 * ad_x_1;
19          else
20              d_z_1 = MTmp5_0 .* ad_x_1;
21          end
22
23          y_0 = y_0 + z_1;
24
25          ssx_0 = numel(y_0);
26          ssy_0 = numel(z_1);
27          if ssx_0 == ssy_0
28              ad_y_0 = ad_y_0 + d_z_1;
29          elseif ssx_0 == 1
30              ad_y_0 = ad_y_0(ones(1,ssy_0), :) + d_z_1;
31          elseif ssy_0 == 1
32              ad_y_0 = ad_y_0 + d_z_1(ones(1,ssx_0), :);
33          else
34              error('internal error in plus');
35          end
36      end
37
38      ad_y_0 = reshape(ad_y_0, [numel(y_0),n_derivs_2]);
```

Figure 3.41: Result of applying AD to code in Figure 3.40

```
1   function [z, d_z] = plus(x, d_x, y, d_y)
2
3     z = x + y;
4
5     if isa(x,'fmad') && isa(y, 'fmad')
6       ssx = numel(x);
7       ssy = numel(y);
8       if ssx == ssy
9         d_z = d_x + d_y;
10      elseif ssx == 1
11        if issparse(d_x)
12          % sparse derivative code omitted for brevity
13        else
14          d_z = d_x(ones(1, ssy), :) + d_y;
15        end  % issparse(d_x)
16      elseif ssy == 1
17        if issparse(d_y)
18            % sparse derivative code omitted for brevity
19        else
20          d_z = d_x + d_y(ones(1, ssx), :);
21        end  % issparse(d_y)
22
23      else
24        error('internal error in plus');
25      end
26    elseif isa(x, 'fmad')
27      if isscalar(x) && ~isscalar (y)
28        d_z = d_x(ones(numel(y), 1), :);
29      else
30        d_z = d_x;
31      end
32    elseif isa(y, 'fmad')
33      if ~isscalar(x) && isscalar (y)
34        d_z = d_y(ones(numel(x), 1), :);
35      else
36        d_z = d_y;
37      end
38    end
```

Figure 3.42: `plus` operation representing `fmad` and `derivvec` `plus` operations

3.5 Code Generation

Once MSAD has applied optimisations like SCCP, copy propagation, constant folding, inlining and DCE, the intermediate SSA-MIR code needs to be converted back to target MATLAB code. This has to be done in two steps, the SSA-MIR code is first turned back into MIR by removing the ϕ nodes inserted as a part of converting the MIR to SSA. The MIR is then converted to MATLAB code by raising the MIR constructs like loops, conditionals and aggregate references like structures and cell arrays which were simplified by the lowering phase.

The first step in converting to target code is to merge the multiple SSA variable copies and remove the SSA ϕ node abstractions. MSAD uses a very efficient and accurate algorithm by Budimlic, et al. [BCH+02] to merge the SSA variable copies originally generated from multiple definitions of the same variable. If no optimisations were applied we could naively rename all the SSA variables originated from the same program variable back to the original variable name and simply delete the ϕ nodes. However many optimisations like copy propagation (which MSAD implements) and CSE (which we intend to implement), can cause two or more SSA copies to be simultaneously *live* in some part of the code, this is termed as a *conflict*. Naively merging the copies will generate the wrong code in this case. A second approach is to simply insert copy assignment statements in the predecessor blocks of the block containing a ϕ node. Whilst this is correct, in practise this approach may introduce an exhorbitant number of copy assignments in the output code. The Fast copy coalescing algorithm [BCH+02] uses *live-range* information to determine if conflicts exist. Conflicting variables retain their unique identity and only necessary copies are inserted. Once all the SSA copies have been handled, the ϕ nodes can be deleted.

The final step is to convert the MIR code to executable MATLAB code. This operation itself is done is several individual steps, as we match CFG patterns in the MIR and map them back to appropriate MATLAB constructs, or match *references*, and explicit *Access* and *Update* operations generated for aggregates like arrays, structures or cell arrays and collapse and convert them

back to MATLAB syntax. The order in which MSAD raises these constructs is:

1. Forward propagate MIR *reference* variables to their uses
2. Raise composite (aggregate) variable operations like array indexing, structure field dereferences and cell array indexing
3. Identify loops and conditionals based on MIR hints and dominator information
4. Raise `for` and `while` loops
5. Raise `if-elseif-else` constructs
6. Raise `m_builtin_end` uses to MATLAB array `end`, MATLAB complete range selector `:`, or use `size` expression to determine its value
7. Forward propagate single use expression temporaries.

The Appendix C and the example 3.41 in the previous section 3.4 show some examples where SSA-MIR has been converted back to executable code. The examples cover loops, conditions and array indexing together with moderately complex use of MATLAB array `end` and `:` syntax.

Chapter 4

Results

Derivatives computed through MSAD, gradients and Jacobians from functions, and Hessians from hand-coded gradients, were tested for correctness and performance on several optimisation problems from MATLAB [Mat11a] and MINPACK [AM91, Len05] test sets, ODE problems from the MATLAB ODE suite [SGT03] and the CWI IVP problem set [MI03], and BVP problems from Shampine, Ketzscher and Forth [SKF05]. The results were compared with MATLAB's finite-differencing routines (`numjac`, `sfd`, `sfdnls`) and MAD. MAD has already been shown to be more efficient than ADMAT in computing derivatives using forward mode on several problems [For06]. In order to test the performance of MSAD in the context of numerical solvers, it was also integrated with the MATLAB solvers `bvp4c` as for MAD [SGT03] and with `ode15s`, `fminunc`, `fmincon`, and `fsolve` using MAD High-Level interfaces [FK04].

These tests were performed using MATLAB R2010b (7.11) on a Ubuntu 10.10 Linux machine with a 2.4GHz Pentium-4 processor and 1024 MB of RAM. Performance comparison of derivative computations is made on the basis of the ratio of CPU time spent in computing the function value and its gradient/Jacobian, to the time required to compute the function value alone. Performance comparison of the complete optimisation/ODE problems is based on the total CPU run-time. The scripts to re-run above tests, and command syntax to invoke MSAD are described in Appendix E.

4.1 ODE problems

4.1.1 Brusselator ODE

Here, we look at results on the `brussode` problem from MATLAB. The Brusselator problem models diffusion in a chemical reaction, and forms a time dependent coupled PDE with two components u and v. The semi-discretisation of the PDE on a 1-D mesh then gives

$$\begin{aligned} u_i' &= 1 + u_i^2 v_i - 4u_i + \alpha(N+1)^2(u_{i-1} - 2u_i + u_{i+1}) \\ v_i' &= 3u_i - u_i^2 v_i + \alpha(N+1)^2(v_{i-1} - 2v_i + v_{i+1}) \end{aligned}$$

This system is typically solved on the time interval $[0, 10]$ with $\alpha = 1/50$ and the initial conditions

$$\left.\begin{aligned} u_i(0) &= 1 + \sin 2\pi x_i \\ v_i(0) &= 3 \end{aligned}\right\} \text{ where } \; x_i = i/(N+1), \;\; i = 1, \cdots, N$$

The system is thus composed of $2N$ equations, where N is the number of grid points. The set of differential equations forms a stiff system that becomes increasingly stiff as the problem size is increased. We employ MATLAB's variable order implicit solver, `ode15s` [SR97] to solve the problem. The implicit Numerical Differentiation Formulae used within the solver computes a solution at any time step by solving a system of nonlinear equations and therefore require a Jacobian of the ODE function. By default this Jacobian is approximated using finite-differences, but a facility is provided to supply an external Jacobian function. We use such a function to supply an AD generated Jacobian to the solver. Further, the Jacobian of the Brusselator problem, shown in Figure 4.1, is sparse with a fixed band diagonal structure of bandwidth five that gets increasingly sparse as the problem size is increased. We therefore use efficient Jacobian compression and sparse derivative propagation techniques for finite-differencing, MAD and MSAD to compute the Jacobian. The Jacobian compression and sparse derivative propagation methods in MSAD are inherited from MAD by virtue of inlining

MAD operations and are the same as described by Forth [For06].

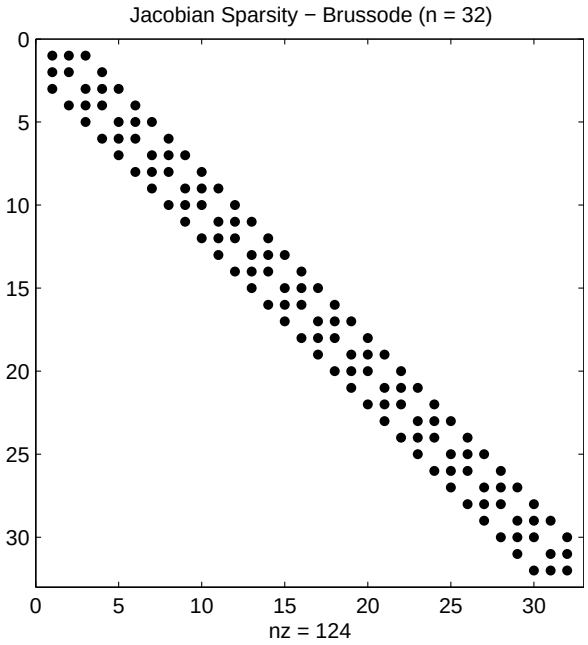

Figure 4.1: Jacobian sparsity pattern for the Brusselator ODE problem (n=32)

Table 4.1 compares the ratio CPU($\mathbf{Jf} + \mathbf{f}$)/CPU($\mathbf{f}$), Jacobian and function computation time to function time alone, for five methods we will subsequently use to supply derivatives to `ode15s`. Function CPU times times are given in Table B.2 of the Appendix B. In comparison to the results ealier in the dissertation [Kha04], where the gains with `msad(comp)` (source transformation and applying Jacobian compression) were 33% to 4% over `fmad(comp)` (overloading with compression), decreasing as the problem size was increased, we see here gains in the range 91% to 59%. Using sparse derivative propagation we see a similar increase in efficiency using complete source transformation, from between 31% to 1% improving to between 84% to 25%. Increasing the problem size beyond $n = 2560$ we see a further decrease in gains, but the gain is still appreciable when using compression, 33% at $n = 10240$. Solving the problem consistently became difficult as the memory limit was approached with problem sizes greater than $n = 11000$, $n = 10240$ was therefore set as a maximum. It should also be noted that

ratios with `fmad` are higher than were presented in the dissertation [Kha04], which is due to the increased overheads in MATLAB 7.x to execute operator overloaded code. The source transformed Jacobian code is now faster than the compressed and vectorised finite-differencing routine, `numjac` from MATLAB for $n \geq 640$.

Table 4.1: Ratio CPU($\mathbf{Jf} + \mathbf{f}$)/CPU($\mathbf{f}$) – Jacobian and function to function CPU time ratio for the Brusselator problem

	CPU($\mathbf{Jf} + \mathbf{f}$)/CPU($\mathbf{f}$) for problem size n								
Method	40	80	160	320	640	1280	2560	5120	10240
`numjac(comp,vect)`	5.3	5.3	5.8	6.5	7.4	8.2	8.8	10.0	10.8
`msad(comp)`	8.0	7.1	6.9	6.7	6.7	6.6	6.8	7.0	8.3
`fmad(comp)`	92.1	74.5	61.6	47.9	34.4	23.5	16.4	13.0	12.4
`msad(sparse)`	16.1	16.1	16.8	18.1	20.8	25.3	33.5	49.1	78.8
`fmad(sparse)`	98.8	83.9	72.1	60.5	50.1	43.7	45.0	56.5	84.1

The total run-time to find a solution to the Brusselator ODE problem using `ode15s` is tabulated in Table 4.2. We observe here that the savings from the source transformed Jacobian code are not reflected proportionately in the total run-time. Despite the 33% decrease in derivative computation time compared to overloading, only 2% decrease in total run-time is seen using `msad(comp)` on the largest problem size. From the solver statistics for this problem we note that the `ode15s` solver requested only two Jacobian computations in the 83 time steps required to determine the complete solution. This is due the Jacobian retention scheme [SR97] adopted in the `ode15s` solver whereby once a Jacobian is computed at any point it is not re-computed until the iteration convergence rate falls below a threshold. Thereafter a decision is made between decreasing the step size of the solver or recomputing the Jacobian.

The entries `msad(comp)` and `fmad(comp)` signify run-times for Jacobian computation using a compressed seed matrix formed by colouring prior to the calls to `ode15s`. If recolouring is to be performed as a part of each computation like in the case of `ode15s` using `numjac`, the entries `msad(comp,recolor)` and `fmad(comp,recolor)` should be read. Although the latter approach

Table 4.2: ODE solution CPU time for the Brusselator problem.

	CPU(ODE solve) (s) for problem size n								
Method	40	80	160	320	640	1280	2560	5120	10240
`numjac(comp,vect)`	0.10	0.12	0.15	0.22	0.36	0.64	1.24	2.55	5.65
`msad(comp)`	0.10	0.12	0.15	0.21	0.34	0.60	1.14	2.30	4.94
`fmad(comp)`	0.15	0.17	0.20	0.26	0.38	0.64	1.19	2.35	5.01
`msad(comp,recolor)`	0.10	0.13	0.16	0.23	0.37	0.68	1.32	2.79	6.39
`fmad(comp,recolor)`	0.15	0.17	0.21	0.28	0.42	0.72	1.37	2.84	6.44
`msad(sparse)`	0.10	0.12	0.15	0.22	0.35	0.62	1.19	2.48	5.61
`fmad(sparse)`	0.15	0.17	0.20	0.27	0.40	0.67	1.25	2.55	5.68

makes a more fair comparison, in problems that require repeated computing of the solution with the problem size remaining the same, the sparsity pattern and the column ordering remain the same and need to be determined only once.

4.1.2 Burgers' ODE

The results presented here are from applying MSAD to solve another ODE problem supplied as a part of the MATLAB ODE solver examples. The Burgers' ODE problem referred to here, is obtained by applying moving mesh method of discretisation, by Huang, et al. [HRR94], to the original PDE formulation of Burgers' equation given by (4.1). Equations (4.2) and (4.3) form the initial conditions for the PDE.

$$\frac{\partial u}{\partial t} = \epsilon \frac{\partial^2 u}{\partial x^2} - \frac{\partial}{\partial x}\left(\frac{u^2}{2}\right), \quad 0 < x < 1,\ t > 0,\ \epsilon = 10^{-4} \tag{4.1}$$

$$u(0,t) = u(1,t) = 0, \qquad t > 0 \tag{4.2}$$

$$u(x,0) = sin(2\pi x) + \frac{1}{2} sin(\pi x), \qquad 0 \leq x \leq 1 \tag{4.3}$$

The discretised ODE forms a *stiff* system of equations [SR97] of the form:

$$M(t,y)\frac{dy}{dt} = f(t,y)$$

where M(t,y) is a *mass matrix* function. The Jacobian of both the ODE and the mass-matrix functions tend to be sparse, and get increasingly sparse as the problem size is increased. Figure 4.2 shows the sparsity pattern of the Jacobian of the ODE function, and Figure 4.3 shows the sparsity pattern of the mass-matrix function in the Burgers' ODE problem. Burgers' ODE is solved using the `odes15s` MATLAB solver which provides facilities to supply the ODE function, the mass matrix, the sparsity pattern of both the ODE and the mass matrix functions.

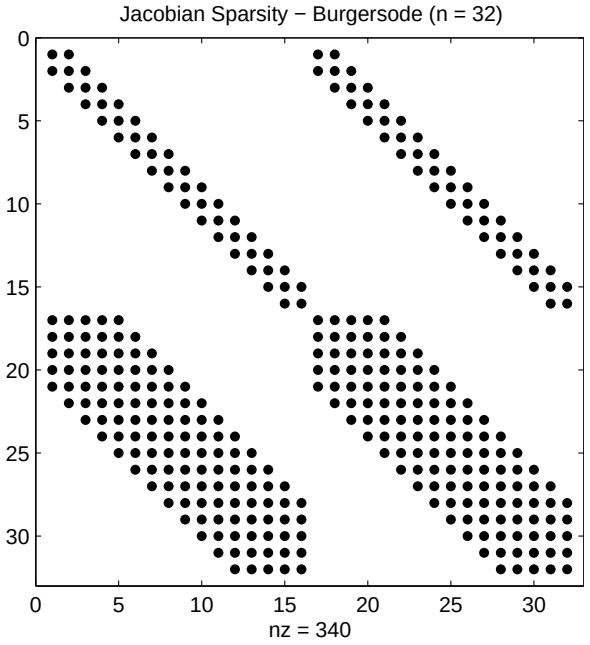

Figure 4.2: Jacobian sparsity pattern for the Burger's ODE problem (n = 32)

Before we look at the performance results of the complete solution using `ode15s`, we will look at Table 4.3 that compares the ratio CPU($\mathbf{Jf} + \mathbf{f}$)/CPU($\mathbf{f}$), Jacobian and function computation time to function time alone, for five methods we will subsequently use to supply derivatives to `ode15s`. The individual function CPU times are given in Table B.3 of the Appendix B. Comparing the results from the three methods `numjac`, `fmad` and `msad` using compressed mode evaluating the Jacobian we see that `msad(comp)` is between 87% to 75% (as the problem size is increased) more efficient in computing the Jaco-

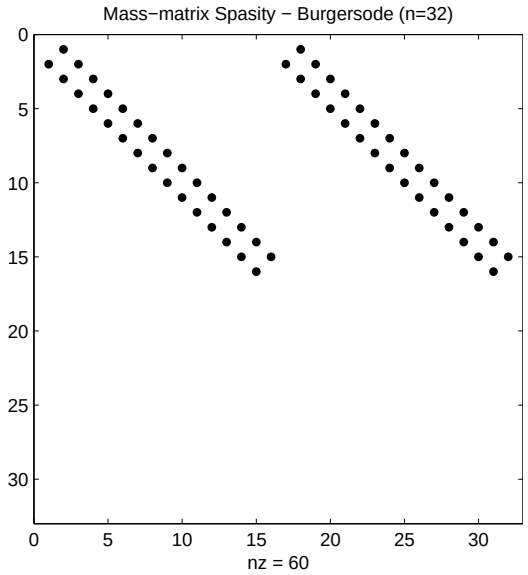

Figure 4.3: Mass-matrix sparsity pattern for the Burger's ODE problem (n = 32)

bian than `fmad(comp)`. Compared to `numjac`, `msad(comp)` gets increasingly efficient as the problem size is increased beyond `n = 128`, being 38% at `n = 512`. Using the sparse mode, `msad(sparse)` is between 77% and 46% more efficient than `mad(sparse)`.

Table 4.3: Ratio CPU($\mathbf{Jf} + \mathbf{f}$)/CPU($\mathbf{f}$) – Jacobian and function to function average CPU time ratio for the Burgers' ODE problem

	CPU($\mathbf{Jf} + \mathbf{f}$)/CPU($\mathbf{f}$) for problem size n					
Method	16	32	64	128	256	512
`numjac(comp)`	20.4	22.8	23.1	23.3	23.7	24.4
`msad(comp)`	26.4	25.3	23.2	18.9	16.5	15.1
`fmad(comp)`	205.3	189.0	163.4	117.5	86.1	60.3
`msad(sparse)`	49.9	50.9	51.8	48.5	50.1	52.6
`fmad(sparse)`	219.5	207.0	186.9	144.8	119.1	98.4

Forth and Ketzscher [FK04] previously described a method to supply MAD generated Jacobians and gradients to MATLAB ODE and optimisation solvers using high level interface functions. We also use these interface functions to supply Jacobian functions generated by MSAD to the ODE

solvers.

The ODE is solved according to the parameters in the original MATLAB example with a relative error tolerance of 10^{-5}, an absolute tolerance of 10^{-4} and over the interval $t = [0, 1]$. Figure 4.4 tabulates the run-time in determining the complete solution of the ODE with the different methods of supplying the ODE and mass-matrix function Jacobian. We obtain a moderate improvement using `msad(comp)` over `fmad(comp)` with savings in run-time between 57% and 13%. Compared to `numjac(comp)`, `msad(comp)` is marginally more efficient as the problem size is increased saving up to 5% in run-time. Like in the previous Brusselator example, the savings in total run-time of the ODE solve is disproportionate to the savings in computing the Jacobian. We attribute this to the Jacobian retention scheme [SR97] explained earlier.

Table 4.4: ODE solution CPU time for the Burger's ODE problem.

	CPU(ODE solve) for problem size n (s)					
Method	16	32	64	128	256	512
`numjac(comp)`	2.07	1.78	1.94	4.42	14.71	75.87
`fmad(comp)`	5.07	5.00	5.23	8.70	21.24	82.86
`fmad(comp,recolor)`	5.07	5.01	5.23	8.72	21.27	82.91
`msad(comp)`	2.16	1.89	2.01	4.34	14.35	72.21
`msad(comp,recolor)`	2.16	1.89	2.00	4.36	14.37	72.25
`fmad(sparse)`	5.23	5.25	5.61	9.60	23.63	89.27
`msad(sparse)`	2.42	2.23	2.47	5.30	16.74	78.50

The Burger's ODE implementation used above was a vectorised version of the original problem, which eliminates explicit loops by using vector indexing and MATLAB array operations. Here we discuss our performance results gathered by differentiating the original non-vectorised version of the Burgers's ODE implementation. The previous versions of MSAD [Kha04, KF06] did not support loops and conditions. In the current implementation we have introduced the support for loops and conditions in order to be able to solve problem not inherently vectorisable or simply larger problems without memory exhaustion due to vectorisation.

Table 4.5 compares the performance of computing the Jacobian of the

Table 4.5: Ratio CPU($\mathbf{Jf} + \mathbf{f}$)/CPU($\mathbf{f}$) – Jacobian and function to function average CPU time ratio for the Burgers' ODE problem (non-vectorised)

	CPU($\mathbf{Jf} + \mathbf{f}$)/CPU($\mathbf{f}$) for problem size n					
Method	16	32	64	128	256	512
`numjac(full)`	37.78	62.00	101.61	203.66	463.11	1074.03
`msad(full)`	185.96	278.00	432.26	729.27	1528.69	4612.50
`fmad(full)`	2359.99	4136.00	6688.70	10704.87	17471.31	40860.57
`numjac(comp)`	37.22	37.86	38.09	39.75	38.58	39.81
`msad(comp)`	196.35	279.61	398.41	571.08	751.96	962.03
`fmad(comp)`	2356.20	4058.25	6539.68	9897.59	13055.11	15765.74
`msad(sparse)`	468.11	777.88	1365.57	2296.38	3907.99	6700.92
`fmad(sparse)`	2576.81	4501.92	7678.68	11695.18	16416.0	22148.14

non-vectorised implementation using `numjac`, `fmad` and `msad`. We observe an order of magnitude improvemnent using `msad` over `fmad` in computing the full Jacobian, using Jacobian compression, and using the sparse mode. We also observe that `numjac` is an order of magnitude more efficient than `msad`. This is counter intuitive considering the results from the vectorised version seen earlier in Table 4.3. Comparing the `numjac` results with `fmad` we observe a difference of two orders of magnitude in the run-time ratio. We believe the inefficiency arises from the `derivvec` array indexing operations, which MSAD inlines. Internally the derivatives corresponding to scalar and array variables, in the `derivvec` class of the MAD package, are stored differently to enhance spatial locality when accessing derivatives values. The current implementation favours array variables, pessimising access of derivatives corresponding to scalar variables. In this non-vectorised Burgers' ODE case, the scalar array accesses inside the `for` loops worsen the run-time efficiency of the code, compared to its vectorised version. We are investigating a fix for this in the MAD package, which will also solve the MSAD regression compared to `numjac`.

4.2 Optimisation problems

MATLAB large-scale optimisation solvers use Trust-Region methods to solve nonlinear minimisation problems by solving quadratic approximations to the original function at each iteration. This involves computing the gradient and Hessian of the objective function at each iteration. The user provides a function that computes the gradient, or gradient and Hessian, at any point in the problem space. If the Hessian is not supplied the solvers internally use a sparse finite differencing routine (`sfd`) to approximate it.

4.2.1 MATLAB large-scale optimisation problems

In Table 4.6 we compare use of MSAD and MAD's `fmad` class to compute derivatives by repeating the large-scale test cases from MATLAB's Optimisation Toolbox [Mat11a] performed in Forth's original work [For06]. The test cases are: `nlsf1a`– sparse Jacobian from vector residual; `brownf`, `tbroyf` – gradient from objective function; `browng`, `tbroyg` – Hessian from hand-coded gradient. Both automatic differentiation tools may use Jacobian/Hessian compression (denoted `cmp`) [GW08, Chap. 8] or sparse storage (denoted `spr`) [GW08, Chap. 7] where appropriate. The only MSAD user directives required were those to specify the active input variables and use of sparse derivative storage. For comparison, we have included MATLAB's finite-difference (`sfd(nls)`) evaluation of the gradient/Jacobian/Hessian and, where available, hand-coding.

Clearly, MSAD yields significant savings compared to `fmad` in like-for-like computation of derivatives for these moderate sized problems ($n \approx 1000$). For compressed derivative computation we get savings of over 50% using `msad(cmp)` and for sparse storage gains of about 30%. Compressed AD (`msad(cmp)`, `fmad(cmp)`) out-performs compressed finite-differencing (`sfd(nls)`). For the gradient problems (`brownf`, `tbroyf`) sparse AD (`msad(spr)`, `fmad(spr)`) is several times faster than `sfd(nls)` because the functions `brownf` and `tbroy` are partially value separable [For06] and the sparse derivative computation may utilise intermediate sparsity whereas finite-differencing cannot. For the `browng` problem `msad(cmp)` outperforms hand-coding due to the use of com-

Table 4.6: Ratio CPU($\nabla \mathbf{f} + \mathbf{f}$)/CPU($\mathbf{f}$) – Jacobian/gradient (including function) to function CPU time ratio for given techniques on MATLAB Optimisation Toolbox large-scale examples. (m, n) gives the number of dependents and independents, $\hat{n}$ the maximum number of non-zero entries in a row of the Jacobian and p the number of colours for compression. Entries marked '–' are not applicable or not available.

	CPU($\nabla \mathbf{f} + \mathbf{f}$)/CPU($\mathbf{f}$) for							
Problem (m,n)	Hand-coded	sfd-(nls)	msad (cmp)	fmad (cmp)	msad (spr)	fmad (spr)	$\hat{n}$	p
nlsf1a(Jac) (1000,1000)	4.4	38.3	6.9	22.5	19.4	35.1	3	3
brownf(grad) (1,1000)	4.6	1064.9	–	–	9.3	13.7	1000	–
browng(Jac) (1000,1000)	5.2	9.5	4.2	8.4	15.3	19.6	3	3
tbroyf(grad) (1,800)	3.8	810.7	–	–	8.8	15.9	800	–
tbroyg(Jac) (800,800)	–	13.8	3.3	10.1	15.8	23.5	6	7

plicated expressions in the naive hand-coding. Details of each problem are given in Table B.1 of the Appendix B.

Table 4.7 lists the total optimisation run-times with derivatives supplied using the methods of Table 4.6. Source transformed derivatives yield substantial savings in the total run-time compared to fmad's overloading approach and run-times are comparable to those using hand-coded derivatives.

4.2.2 MINPACK 2-D Ginzburg-Landau problem

The 2-D Ginzburg-Landau problem (GL2) from the MINPACK-2 problem set [ACMX92, Len05] is an unconstrained minimisation problem that minimises the Gibbs free energy in Ginzburg-Landau superconductivity equations. The problem is discretised over an $(n_x + 2) \times (n_y + 2)$ grid $\mathbf{z}_{i,j} = (x_{i,j}, y_{i,j})$ and takes the form:

$$min \left\{ \sum \left(f_{i,j}^{(1)}(\mathbf{v}) + f_{i,j}^{(2)}(\mathbf{v}, \mathbf{a}) : \mathbf{v} \in \mathbb{C}^m, \mathbf{a} \in \mathbb{R}^{2m} \right) \right\}$$

Table 4.7: Averaged CPU time for optimisation of the large-scale examples from the MATLAB Optimisation Toolbox with derivatives supplied using given techniques. Entries marked '–' are not applicable or not available.

Problem	Optimisation CPU time (s) for					
	`Hand-coded`	`sfd-(nls)`	`msad (cmp)`	`fmad (cmp)`	`msad (spr)`	`fmad (spr)`
`nlsf1a`	0.16	0.36	0.17	0.31	0.20	0.35
`brownf`	0.56	–	–	–	0.7	1.25
`browng`	0.29	0.56	0.23	0.41	0.46	0.64
`tbroyf`	0.72	–	–	–	1.29	2.89
`tbroyg`	–	0.76	0.20	0.48	0.55	0.86

where $f_{i,j}^{(1)}$ is a function of $\mathbf{v}$, the order parameter at point $\mathbf{z}_{i,j}$, and $f_{i,j}^{(2)}$ is a function of $\mathbf{v}$ and $\mathbf{a}$, the vector potential at points $\mathbf{z}_{i,j}, \mathbf{z}_{i+1,j}, \mathbf{z}_{i,j+1}$. The minimisation is carried out on the interior points of the grid and the problem size is then $n = 4n_x n_y$. We employ MATLAB's `fminunc` solver to perform a large-scale minimisation starting with the standard MINPACK provided initial point for the problem. The Hessian function supplied to `fminunc` is derived as earlier, treating the Hessian as a Jacobian ($\mathbf{Jg}$) of a hand-coded gradient function denoted here by $\mathbf{g}$. The Hessian matrix, shown in Figure 4.4, gets increasingly sparse as the problem size is increased. We therefore apply compression techniques or sparse derivative propagation to compute this Hessian matrix.

Table 4.8 gives the derivative computation ratio $\text{CPU}(\mathbf{Jg} + \mathbf{g})/\text{CPU}(\mathbf{g})$ of the GL2 problem as the problem size is increased. The Jacobian computation using `msad(comp)` is nearly 80% more efficient than `fmad(comp)` for smaller problem sizes. As the problem size is increased the core derivative computation time increases and the overheads from overloading relatively decrease. But the overheads from operator overloading can clearly be seen with this moderately large program of about 300 lines of code. Even at problem sizes as large as $n = 65536$, `msad(comp)` shows an improvement of $\approx 45\%$ (nearly twice as fast as overloading). Using sparse derivatives, `msad(sparse)` and `fmad(sparse)`, shows a similar trend but with a smaller relative improvement, 77% down to 21%. A considerable portion of the computation

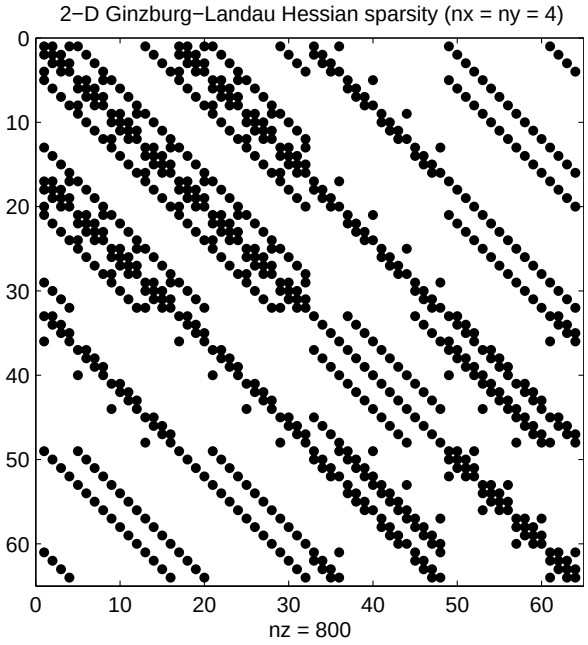

Figure 4.4: Hessian sparsity pattern of the 2-D Ginzburg-Landau problem (n=64)

time is spent in manipulating MATLAB's internal `sparse` data structures that hold the derivative values, diminishing improvements obtained through source transformation.

Table 4.8: Ratio CPU($\mathbf{Jg} + \mathbf{g}$)/CPU($\mathbf{g}$) – Hessian and gradient to gradient function CPU time ratio for the MINPACK 2-D Ginzburg-Landau problem

	CPU($\mathbf{Jg} + \mathbf{g}$)/CPU($\mathbf{g}$) for problem size n					
Method	64	256	1024	4096	16384	65536
`msad(comp)`	24.72	23.69	21.84	23.31	37.95	52.16
`fmad(comp)`	115.32	105.89	89.57	72.82	72.11	90.47
`msad(sparse)`	28.11	29.18	35.02	52.63	88.97	177.10
`fmad(sparse)`	122.80	113.84	107.73	108.72	126.45	222.81
`#colours` p	20	23	25	24	25	25

The time required to compute a local minimum, with different AD methods supplying the Hessian and with increasing problem sizes, is shown in Table 4.9. The decrease in overall computation time due to the decrease in Hessian computation time by using `msad` can be seen. We also observe that

this decrease in the overall computation time with increasing problem size is not in proportion to the decrease in derivative computation time. Although the number of *Newton iterations* required to compute a solution does not vary significantly, the number of *conjugate-gradient* iterations required within the solver to form a two-dimensional trust-region subproblem for optimisation increases as the problem size is increased. This increase in computational load diminishes the relative gains from the speedup in derivative computation. Bouaricha, Moreé and Wu's report [BMW97] shows the distribution of computational load using a Newton method on various MINPACK problems. The GL2 problem has a larger percentage of computation time devoted to the CG iterations (33%) than the Hessian matrix computation (19%), and this concurs with our results. The relative proportion of the Hessian computation time on the other problems of the MINPACK set also suggest that the overall performance improvement due to speedup in derivative computation will be higher on these problems. The MATLAB implemetation of the GL2 problem was obtained from Lenton's project [Len05]. The results discussed above were obtained using the vectorised form of the code.

Table 4.9: Solution CPU time for the MINPACK 2-D Ginzburg-Landau problem using MATLAB large scale optimisation solver `fminunc`. The problem size (number of variables) is $n = 4n_x n_y$ with $n_x = n_y$

	Problem size n (n variables)					
Method	64	256	1024	4096	16384	
`msad(comp)`	0.74	0.59	1.34	6.95	29.62	
`fmad(comp)`	4.45	2.78	3.79	10.71	38.70	
`msad(sparse)`	1.23	1.14	2.87	13.05	61.93	CPU time (s)
`fmad(sparse)`	4.79	3.32	5.41	17.05	73.83	(complete solution)
`sfd`	1.41	1.29	3.60	19.91	216.30	
Iteration						
`Newton steps`	21	12	13	15	12	Iteration
`CG steps`	200	199	423	835	1195	count

4.2.3 Other smaller dimension MINPACK problems

Performance comparisons for smaller dimensional optimisation problems from the MINPACK problem set are presented here. These problems are solved using MATLAB's toolbox functions `fsolve`, for solving system of non-linear equations, and `lsqnonlin`, for solving a system of non-linear equations in a least-squares sense. Both methods utilise the Jacobian of the system of equations. In Table 4.10 we compare the performance of computing the Jacobian of the system of equations using finite-differencing `(sfd)` and AD `(msad, mad)`. Speedup of `msad` over `fmad` for these small cases is clearly seen.

Table 4.10: Ratio CPU($\mathbf{Jf} + \mathbf{f}$)/CPU($\mathbf{f}$) – Jacobian and function to function CPU time ratio for smaller dimension Nonlinear least-squares and Nonlinear system of equation problems from the MINPACK set

		Ratio CPU($\mathbf{Jf} + \mathbf{f}$)/CPU($\mathbf{f}$)		
Problem	n	`sfd`	`msad(full)`	`mad(full)`
Human heart dipole	8	23.12	53.08	737.16
Propane combustion	11	22.22	35.03	394.29
Coating thickness stand.	134	256.28	49.65	107.87

Table 4.11 gives the complete CPU run-time of the optimisation problem together with the number of iterations used during the optimisation. The Propane combustion problem is solved here using both optimisation methods to convergence. Savings in complete optimisation time using `msad` to compute derivatives, over `fmad` are clearly seen. Although `sfd` computes derivatives on two of these problems more efficiently than `msad`, the total optimisation times are almost the same. In the second case we observe the number of iterations used by the solver using AD is smaller than that using finite-differencing. The higher accuracy with AD generated derivatives may explain this result.

Performance comparisons on computing derivatives using various methods on several other ODE and optimisation problems are available in the Appendix A.

Table 4.11: Solution CPU time for smaller dimension Nonlinear least-squares and Nonlinear system of equation problems from the MINPACK set

Run-time (s) for solution to Non-linear system & Least squares problems						
Problem	Solver	`sfd`	#Iter	`msad (full)`	`mad (full)`	#Iter
Human heart dipole	`fsolve`	0.27	62	0.31	2.08	62
Propane combustion	`lsqnonlin`	0.26	103	0.26	2.70	96
Propane combustion	`fsolve`	0.11	21	0.12	0.75	21
Coating thickness stand.	`lsqnonlin`	2.49	30	0.70	1.38	30

4.3 MSAD support for `bvp4cAD`

MATLAB's `bvp4c` routine [KS01] solves a general two-point boundary value problem involving ordinary differential equations, and allows for unknown parameters p in $dy/dx = f(x, y, p)$ and singularities in the solution. The solver transforms the problem into a system of nonlinear equations via a 3-stage Lobatto collocation formula and applies Newton's method to find a solution. MSAD has been successfully tested to run from within a modified MATLAB `bvp4cAD` solver [SKF05] to provide the AD generated Jacobian $\partial \mathbf{f}(x_i, \mathbf{y}_i, \mathbf{p})/\partial \mathbf{y}_i$ and, in case of unknown parameters, this includes $\partial \mathbf{f}(x_i, \mathbf{y}_i, \mathbf{p})/\partial \mathbf{p}$. A similar Jacobian for the boundary condition residual function is also formed. This is done by invoking MSAD on the two user specified functions, `odefun` and `bcfun`, from within the solver. The `bvp4cAD` routine internally calls MSAD, although MSAD carries out actual code augmentation only the first time a problem is presented or when the input problem file is changed. MATLAB uses a native Java Virtual Machine (JVM) in which the MATLAB interpreter is run. This reduces the overhead of invoking external Java code using the JVM by several times as compared to using a `system` command to invoke conventional compiled binaries. In the early Java version of MSAD [KF05] we called MSAD on every invocation of `bvp4cAD` and compared the timestamps within MSAD. The most recent version of MSAD is written in C++ and compiled to a native binary and therefore needs to be invoked using the `system` command. However, the trivial task

of comparing the time stamps of the test input and the differentiated output is now simply done within `bvp4cAD` by comparing the timestamps returned by the MATLAB `dir` command. This circumvents the overhead of calling MSAD on every invocation of `bvp4cAD`.

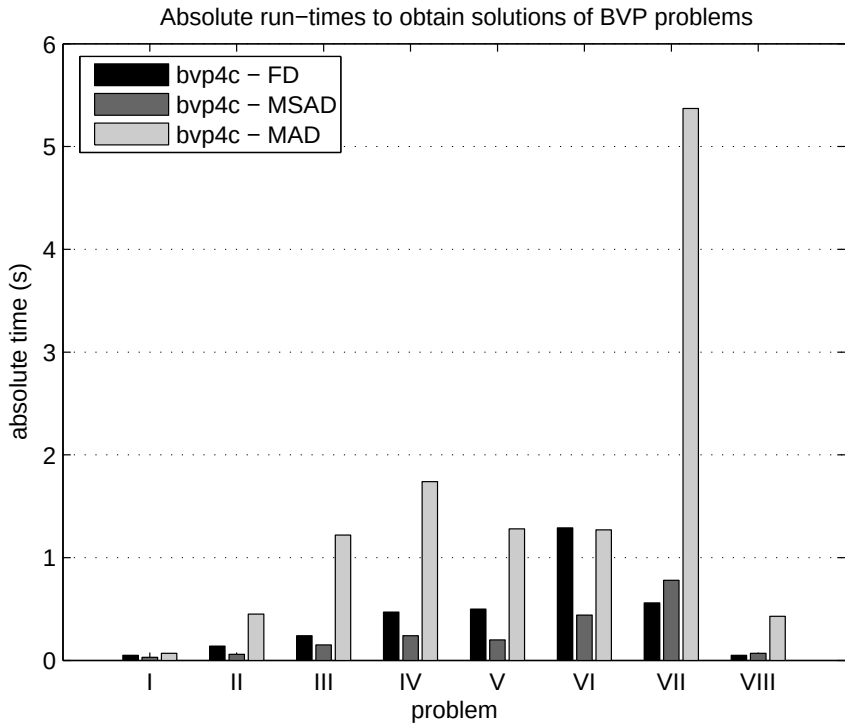

Figure 4.5: Absolute run-times for complete solutions to boundary value problems using `bvp4c` and `bvp4cAD`

Figure 4.5 compares the total run-time to obtain a solution with the standard `bvp4c` that uses finite-differences to approximate the Jacobian of the function and residual, with `bvp4cAD` using MSAD and MAD to provide an AD generated Jacobian. The run-times are tabulated in Table B.5 of the Appendix B. On six of the eight problems tested previously by Shampine, Gladwell and Thompson [SGT03] we can see the performance using MSAD is better than finite-differencing. On the other two cases, the performance is comparable. Using MSAD is also significantly faster than MAD.

4.4 Experiments with CSE

In Chapter 1, Section 1.4.3 we mentioned some of the benefits of using source transformation and the CSE optimisation in particular. Table 1.6 compared the run-time of the original program in Figure 1.1 with the optimised program in Figure 1.2 where we observed on average 20% improvement in performance of the function run-time alone. Here we demonstrate the effect of the CSE optimisation on AD augmented code.

Table 4.12: Ratio CPU($\mathbf{Jg} + \mathbf{g}$) (s) – CPU time for AD derived Hessian from gradient function of the Brown problem averaged over 100 runs

	CPU($\mathbf{Jg} + \mathbf{g}$) (s) for problem size n							
Method	256	512	1024	2048	4096	8192	16384	32768
`hand coded`	0.26	0.48	0.91	1.79	3.57	7.29	15.85	34.60
`sfd`	0.50	0.98	2.25	5.94	18.01	60.63	226.38	1071.40
`msad(comp)`	0.33	0.55	1.04	2.12	4.33	9.67	20.51	41.73
`msad(comp,cse)`	0.19	0.31	0.56	1.15	2.40	5.65	11.98	24.13
`fmad(comp)`	1.31	1.54	2.02	3.03	5.43	12.55	28.56	58.47
`fmad(comp,cse)`	0.88	1.01	1.29	1.93	3.35	8.05	17.76	36.53
`msad(sparse)`	1.00	1.83	3.75	8.46	21.43	61.92	201.38	787.73
`msad(sparse,cse)`	0.71	1.32	2.78	6.62	17.85	54.64	186.51	715.67
`fmad(sparse)`	1.96	2.80	4.72	9.50	22.78	64.05	205.13	795.13
`fmad(sparse,cse)`	1.38	2.02	3.52	7.45	19.03	57.20	192.51	761.60

The performance figures in Table 4.12 compare the differentiated versions of the original program in Figure 1.1 with the optimised program in Figure 1.2. Performance figures from MAD and finite-differencing routine `sfd` are also presented for reference. Comparing the source transformation methods `msad(comp,cse)` with `msad(comp)`, we observe a consistent 42% improvement with the CSE optimisation across all problem sizes. A similar decrease occurs when using operator overloading for differentiation of the optimised code. Comparing `fmad(comp,cse)` and `fmad(comp)`, we see that the benefits from removing the common indexing operations are more pronounced here. It can also be observed that the optimised routine is faster than

the hand-coded Hessian routine in this case. It should be noted that the CSE optimisation here is applied only to the original function. Because we intend to introduce CSE optimisation into MSAD as a post differentiation optimisation, technically we could compare `msad(comp,cse)` with `fmad(comp)`, the un-optimised overloaded code. In such a case we would obtain performance improvements in excess of 94% $\rightarrow$ 59% in computing the derivatives, asserting the benefits of such high level optimisations during MATLAB source transformation AD.

MSAD does not implement the CSE optimisation at present, but we have all the necessary infrastructure in place to implement both local and global common subexpression elimination and many other optimisations that we believe will benefit AD.

4.5 MSAD performance

So far we have seen the performance of the augmented code generated from MSAD and the effect of increased efficiency in computing the derivatives on ODE and optimisation solvers. Here we discuss the offline overhead of MSAD in processing the input program to generate augmented AD code itself.

Table 4.13: MSAD AD augmentation overhead

problem	cpu time (s)	#operators augmented	#input lines of code	#output lines of code
fdaer	0.64	13	8	67
gbrown	2.55	68	14	944
fdgl2nonvec	5.30	185	56	642
fburgersodenonvec	12.70	346	64	1385
fburgersode	13.55	334	54	2236
gdgl2nonvec	50.20	1432	293	5961

MSAD was compiled with the GNU `g++` compiler from `GCC 4.4.3` with the `-O2` optimisation level for timing purposes, as would be the case with the deployed version. We usually build in debug mode with `-O0 -g` to aid

development. The Table 4.13 gives a few examples of the typical run-times of MSAD on examples seen earlier to generate augmented AD code. Column two of Table 4.13 lists the CPU time collected using the *time* Linux utility. The third column lists the number of operators that MSAD augmented during the processing. Because the number of operators in a program is usually not used to present the complexity of programs, in columns four and five we list the number of lines of code (formatted to 80 columns) of the input program to MSAD, and the output generated from MSAD respectively. We can see that the run-time is linearly proportional to the number of operators in the input program. This is as expected becase each operator in the input program that needs to be augmented needs specialising and inlining. In Chapter 3, Section 3.4 we showed how the MSAD recursively calls parsing, optimising and inlining passes to specialise `fmad` and `derivvec` operations for each operation involving active operands.

Table 4.14: MSAD compile-time overhead

problem	cpu time (s)
fdaer	0.124
gbrown	0.148
fdgl2nonvec	0.308
fburgersodenonvec	0.344
fburgersode	0.300
gdgl2nonvec	1.212

Table 4.14 shows the CPU-time in processing the same problems as in Table 4.13 but without the AD augmentation (including analysis and optimisations). We observe that the run-time is almost an order of magnitude smaller, indicating that the core MSAD operations are efficient. Profiling the MSAD code using compiler instrumentation `-pg` option in `gcc` we discovered close to 40% of the run-time is spent in scanning, parsing and converting the AST to IR. Because augmenting each operator currently involves repeatedly parsing the M-files for each overloaded operation, we can gain significantly from caching the MIR for files already processed the first time. We only need

to create copies in memory on subsequent re-use of the same operation. We believe this will make a significant impact on improving the performance in future implementations.

Chapter 5

Conclusions and Future work

In this thesis we have detailed our implementation of MSAD, a source transformation tool for Algorithmic Differentiation (AD) of MATLAB.

Chapter 1 introduced the concept of AD and discussed the relevance of AD in MATLAB. We discussed the operator overloading and source transformation methods used to implement AD in MATLAB and the performance issues encountered with them. We also showed some generic performance pitfalls in MATLAB arising from programming constructs such as array indexing, and discussed interpretation and overloading overheads. This motivates our implementation of source transformation AD and independently the need for compiler optimisations in a MATLAB source transformation framework. Finally we set up goals for MSAD and some basic requirements from our MSAD tool.

Chapter 2 introduced some basic compiler concepts together with the more advanced concepts we have used such as the intermediate representation (IR), control- and data-flow analysis, use of lattices in data-flow analysis. We also introduced the *Single Static Assignment* (SSA) form of the IR which simplifies analyses and optimisations. We looked at some optimisations relevant to AD like constant and copy propagation, dead code elimination and inlining with examples from MSAD.

In Chapter 3 we looked very closely at implementation details of MSAD and the choice of algorithms used in our MSAD compiler framework. We

started by introducing ANTLR and showed some examples of using lexer and parser rules to parse MATLAB and generate an AST. We discussed the symbol resolution algorithm in MSAD in detail which is the key to disambiguate common syntax for array indexing and function calls. We looked at how MSAD lowers complex MATLAB constructs like loops, conditions and structure, array aggregate constructs to our own medium-level intermediate representation (MIR). We also showed how this IR is converted to SSA that forms the backbone to MSAD's analyses and optimisation framework. We demonstrated a novel use of the *Sparse conditional constant propagation* (SCCP) algorithm and composite attribute-lattices to infer properties of MATLAB variables such as class, sparsity, complexness, rank, dimensions and value. We showed how the entire infrastructure is re-used to parse the overloaded operations from MAD to SSA-MIR, specialise using SCCP, and prune the dead code after specialisation using Dead Code Elimination (DCE). Finally, surplus SSA generated copies are reclaimed and the IR converted back to target MATLAB code.

The results in Chapter 4 showed how MSAD was successfully applied to compute the first order derivatives of various ODE, Optimisation and BVP problems, both large and small. Comparing the ratio of time to compute the Jacobian to the original function on the Brusselator ODE problem showed MSAD is on average 65% more efficient than MAD with the compressed mode of Jacobian computation on large problem sizes, and close to 30% with the sparse mode. MSAD is also 23% more efficient than `numjac` on large problem sizes with the compressed mode. Similar analysis of the Burgers' ODE problem shows MSAD is on average 80% more efficient than MAD using the compressed mode, and close to 50% with the sparse mode. MSAD is also 38% more efficient compared to `numjac` on large problem sizes using the compressed mode. Comparing the overall time in computing the solution of the ODE using MSAD and MAD supplied derivatives, showed on average 30% improvement in performance on the Burgers' ODE problem. Similar improvements were also obtained on MINPACK optimisation problems like 2-D Ginzburg-Landau problem. On smaller sized problems MSAD is almost always an order of magnitude better in performance than MAD in computing

derivatives, and comparable to finite-differencing. This clearly demonstrates that MSAD successfully removes overheads from operator overloading making code more amenable to MATLAB just-in-time compilation (JIT) and hence more efficient. In the process of differentiating non-vectorised codes we have identified a performance issue in the MAD package concerning storage and access of derivatives associated with scalar variables. We are investigating this problem following which it will be fixed in the MAD package and consequently available through MSAD.

Throughout the process of design and implementation we have consciously made an attempt to keep the implementation generic and modular so as not to be biased by the final application. Yet, we have been pragmatic in the choice of optimisations and transformations to implement in the given time constraints. Through this implementation we have demonstrated that the process of specialising and inlining is feasible, and in the process also built an extensible infrastructure to allow incorporation of new optimisations and algorithms requiring minimal or no change to the current infrastructure. We have carefully chosen efficient algorithms for the most expensive tasks in the MSAD infrastructure to ensure a reasonable run-time of the MSAD tool itself. As a software, MSAD is written in C++ and comprises nearly $150,000$ lines of code, including generated code from the scanner, parser and tree-parser specifications for various analyses.

In line with the basic requirements set at the start in Chapter 1, Section 1.5 we have demonstrated the following:

1. *Readability of augmented code* - From the examples of augmented code in Chapter 3, Sections 3.4, and the Appendix C we see that MSAD generates sufficiently readable MATLAB code. MSAD also provides output code formatting options to control the length of the lines, output of white-spaces, etc. We are considering adding comments to the output, to hint where AD augmented code was inserted. We are also considering renaming all the automatically generated IR names before the final code is output in order to generate more uniform smaller names to further increase readability.

2. *Correctness of augmented code* - On all the tests discussed in Chapter 4, MSAD generated derivative results have been compared to those from the parent MAD implementation for numerical accuracy. The little discrepancies that we have observed are in line with those from the differing floating point round-off errors we would expect from constant folding optimisations. All of the constant folding operations in MSAD's constant folding library are currently based on C++ *double* precision arithmetic provided by the GNU `libsdtc++` libraries. In future we may consider using more precise GMP and MPFR [FHL+07] implementations to fold constants.

3. *Correctness of MSAD optimised code* - In the black-box testing discussed in Chapter 4 and several large programs like `ode15s` from MATLAB itself, along with unit tests for most commonly used MATLAB syntax, we found no numerical errors generated by MSAD optimised programs. For testing `ode15s`, we compared the results of solving Burgers' ODE problem in Chapter 4 with the original and the MSAD processed version of `ode15s` and found no errors. We are still stabilising coverage of syntax like MATLAB's `switch cases` and exception handling via `try-catch` which were not tested thoroughly.

4. *Reasonably small run-time* - In addition to showing improvements from MSAD generated AD code, Chapter 4 showed that the run-time of the tool itself, when generating AD code, is reasonably small. This run-time scales linearly as a function of the number of operations in the input program that need to be augmented. We expect a significant improvement once we implement caching of the MIR for operations commonly inlined. This will be done as part of improvements to the inter-procedural analysis which will build and use a call graph to determine functions to be inlined and those that are freqently used. When purely optimising MATLAB code, MSAD is seen to be very efficient owing to its use of efficient algorithms and lean IR.

5. *Self-contained* - With a view to retaining maintainability and avoid

feature creep we have avoided dependending on any large third party software in our work. The only dependency is on ANTLR 2.7 which is free to modify and or use. During development we have used the standard GNU Compiler Collection (v4.3 - v4.4) and run-time on both Cygwin and Ubuntu-linux environments. However, MSAD can theoretically be compiled and used on any other platform.

Finally, we trust that the infrastructure we have produced will enable continued development of source transformation techniques for MATLAB, not just for first order derivatives but also for other forward propagated entities such as higher derivatives [GW08, Ch.13] or intervals [Rum99]. Of course a further aim would be to enable reverse mode AD, this would involve significant further work due to the significant requirement for control-flow reversal.

Bibliography

[ACMX92] Brett M. Averick, Richard G. Carter, Jorge J. Moré, and Guo-Liang Xue. The MINPACK-2 test problem collection. Preprint MCS–P153–0692, ANL/MCS–TM–150, Rev. 1, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1992. See `ftp://ftp.mcs.anl.gov/pub/MINPACK-2/tprobs/P153.ps.Z`.

[AFSS00] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. A comparative study of static and profile-based heuristics for inlining. *SIGPLAN Not.*, 35:52–64, January 2000.

[Aho90] Alfred V. Aho. *Algorithms for finding patterns in strings*, pages 255–300. MIT Press, Cambridge, MA, USA, 1990.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Adison-Wesley, 1974.

[AM91] Brett M. Averick and Jorge J. Moré. User guide for the MINPACK-2 test problem collection. Technical Memorandum ANL/MCS-TM-157, Argonne National Laboratory, Argonne, Ill., 1991. Also issued as Preprint 91-101 of the Army High Performance Computing Research Center at the University of Minnesota.

[AP02] George Almási and David Padua. MaJIC: compiling MATLAB for speed and responsiveness. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 294–303, New York, NY, USA, 2002. ACM Press.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Boston, USA, 1986.

[BBL+02] C.H. Bischof, H.M. Bücker, B. Lang, A. Rasch, and A. Vehreschild. Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages p. 65–72. IEEE Computer Society, 2002.

[BBV05] Christian H. Bischof, H. Martin Bücker, and Andre Vehreschild. A macro language for derivative definition in ADiMat. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, Lecture Notes in Computational Science and Engineering, pages 181–188. Springer, 2005.

[BCC+92] Christian H. Bischof, Alan Carle, George F. Corliss, Andreas Griewank, and Paul D. Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1:11–29, 1992.

[BCH+02] Zoran Budimlic, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. Fast copy coalescing and live-range identification. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 25–32, New York, NY, USA, 2002. ACM.

[BMS+96] Martin Berz, Kyoko Makino, Khodr Shamseddine, Georg H. Hoffstätter, and Weishi Wan. COSY INFINITY and its applications to nonlinear dynamics. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 363–365. SIAM, Philadelphia, Penn., 1996.

[BMW97] A. Bouaricha, Jorge J. Moré, and Zhijun Wu. Newton's method for large-scale optimization. Preprint MCS-P635-0197, Argonne National Laboratory, Argonne, Illinois, 1997.

[BPK96] Christian Bischof, Gordon Pusch, and Ralf Knoesel. Sensitivity analysis of the MM5 weather model using automatic differentiation. *Computers in Physics*, 10(6):605–612, 1996.

[BRM97] Christian H. Bischof, Lucas Roh, and Andrew Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software – Practice and Experience*, 27(12):1427–1456, 1997. See `www-fp.mcs.anl.gov/adic/`.

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 55–66, New York, NY, USA, 1991. ACM.

[CFR+89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 25–35, New York, NY, USA, 1989. ACM.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CG94] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 397–408, Portland, Oregon, 1994.

[CHK04] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. Iterative data-flow analysis, revisited. Technical report, Rice University, 2004.

[CV98a] Thomas F. Coleman and Arun Verma. ADMAT: An automatic differentiation toolbox for MATLAB. Technical report, Computer Science Department, Cornell University, 1998.

[CV98b] Thomas F. Coleman and Arun Verma. The efficient computation of sparse Jacobian matrices using automatic differentiation. *SIAM J. Sci. Comput.*, 19(4):1210–1233, 1998.

[CV00] Thomas F. Coleman and Arun Verma. ADMIT-1: Automatic differentiation and MATLAB interface toolbox. *ACM Transactions on Mathematical Software*, 26(1):150–175, 2000.

[DS95] C. Donnelly and R. Stallman. *Bison: the YACC-compatible parser generator*. Free Software Foundation, 1995.

[Eat11] J.W. Eaton. GNU Octave – a high-level language for numerical computations. *http://www.octave.org*, 2011.

[ELC03] Daniel Elphick, Michael Leuschel, and Simon Cox. Partial evaluation of MATLAB. In *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, pages 344–363, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[Fas03] FastOpt. *Transformation of Algorithms in Fortran, Manual, Draft Version, TAF Version 1.6*, Nov. 2003. See `http://www.FastOpt.com/taf`.

[FE04] Shaun A Forth and Marcus M. Edvall. *User Guide for MAD - MATLAB Automatic Differentiation Toolbox TOMLAB/MAD, Version 1.1 The Forward Mode*. TOMLAB Optimisation Inc., 855 Beech St 12, San Diego, CA 92101, USA, Jan 2004. See `http://tomlab.biz/products/mad`.

[FHL+07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13:1–13:15, June 2007.

[FK04] Shaun A. Forth and Robert Ketzscher. High-level interfaces for the MAD (Matlab Automatic Differentiation) package. In P. Neittaanmäki, T. Rossi, S. Korotov, E. Oñate, J. Périaux, and D. Knörzer, editors, *4th European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS)*, volume 2. University of Jyväskylä, Department of Mathematical Information Technology, Finland, Jul 24–28 2004. ISBN 951-39-1869-6.

[For06] Shaun A. Forth. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Transactions on Mathematical Software*, 32(2):195–222, June 2006.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, 1987.

[FTPR04] Shaun A. Forth, Mohamed Tadjouddine, John D. Pryce, and John K. Reid. Jacobian code generated by source transformation and vertex elimination can be as efficient as hand-coding. *ACM Trans. Math. Softw.*, 30(3):266 – 299, Sep. 2004.

[GBJL86] D. Grune, H. Bal, C. Jacobs, and K. Langendoen. *Modern Compiler Design*. Computer Science. Wiley, Chichester, NY, 1986.

[GJU96] Andreas Griewank, David Juedes, and Jean Utke. Algorithm 755: ADOL–C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw.*, 22(2):131–167, 1996.

[GMS92] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, Jan. 1992.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23:5–48, March 1991.

[Gri93] A. Griewank. Some bounds on the complexity of gradients, Jacobians, and Hessians. In *Complexity in Nonlinear Optimization*, pages 128–161. World Scientific Publishers, 1993.

[GW08] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.

[HGP03] Laurent Hascoet, Rose-Marie Greborio, and Valerie Pascual. Computing adjoints by automatic differentiation with TAPENADE. In *Problemes non-lineaires appliques*. Springer, 2003.

[HHG05] Patrick Heimbach, Chris Hill, and Ralf Giering. An efficient exact adjoint of the parallel mit general circulation model, generated via automatic differentiation. *Future Gener. Comput. Syst.*, 21(8):1356–1371, October 2005.

[HR92] David R. Hill and Lawrence C. Rich. Automatic differentiation in MATLAB. *Applied Numerical Mathematics*, 9:33–43, 1992.

[HRR94] Weizhang Huang, Yuhe Ren, and Robert D. Russell. Moving mesh methods based on moving mesh partial differential equations. *J. Comput. Phys*, 113:279–290, 1994.

[HVD03] Laurent Hascoët, Mariano Vázquez, and Alain Dervieux. Automatic differentiation for optimum design, applied to sonic boom reduction. In V. Kumar, M. L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, editors, *Computational Science and Its Applications – ICCSA 2003, Proceedings of the International Conference on Computational Science and its Applications, Montreal, Canada, May 18–21, 2003. Part II*, volume 2668 of *Lecture Notes in Computer Science*, pages 85–94, Berlin, 2003. Springer.

[HW76] G. Hall and J.M. Watt. *Modern numerical methods for ordinary differential equations.* Oxford Press, Bristol, 1976.

[JB06] Pramod G. Joisha and Prithviraj Banerjee. An algebraic array shape inference system for MATLAB. *ACM Trans. Program. Lang. Syst.*, 28(5):848–907, 2006.

[Joh75] S.C. Johnson. Yacc: Yet another compiler compiler. Technical report, Bell Laboratories, Murray Hill, NJ 07974, 1975. Computing Science Technical Report No. 32.

[KF05] R.V. Kharche and S.A. Forth. Source transformation for MATLAB automatic differentiation. Applied Mathematics and Operational Research Report AMOR 2005/1, Cranfield University (Shrivenham Campus), Swindon SN6 8LA, UK, December 2005.

[KF06] Rahul V. Kharche and Shaun A. Forth. Source transformation for MATLAB automatic differentiation. In Vassil N. Alexandrov,

Geert Dick van Albada, Peter M.A. Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *Lecture Notes in Computer Science*, pages 558–565, Heidelberg, 2006. Springer.

[Kha04] Rahul V. Kharche. Source transformation for automatic differentiation in MATLAB. Master's thesis, Cranfield University (Shrivenham Campus), Applied Mathematics & Operational Research Group, Engineering Systems Department, RMCS Shrivenham, Swindon SN6 8LA, UK, Aug. 2004.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Adison-Wesley, 1997.

[KS01] Jacek Kierzenka and Lawrence F. Shampine. A BVP solver based on residual control and the MATLAB PSE. *ACM Trans. Math. Softw.*, 27(3):299–316, 2001.

[KWBV05] M. Kalkuhl, W. Wiechert, H. M. Bücker, and A. Vehreschild. High precision satellite orbit simulation: A test bench for automatic differentiation in MATLAB. In F. Hülsemann, M. Kowarschik, and U. Rüde, editors, *Proceedings of the Eighteenth Symposium on Simulation Techniques, ASIM 2005, Erlangen, September 12–15*, number 15 in Frontiers in Simulation, pages 428–433, Erlangen, 2005. SCS Publishing House.

[LEJ79] Thomas Lengauer, Robert Endre, and Tar Jan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1:121–141, 1979.

[Len05] Katharine Lenton. An efficient, validated implementation of the MINPACK-2 test problem collection in MATLAB. Master's thesis, Cranfield University (Shrivenham Campus), Applied Mathematics & Operational Research Group, Engineering Systems Department, RMCS Shrivenham, Swindon SN6 8LA, UK, 2005.

[LLV11] The LLVM Team. *LLVM Compiler Collection Internals*. University of Illinois Urbana-Champaign, Illinois, April 2011. LLVM version 2.9 *http://llvm.org/releases/2.9/docs/index.html*.

[LS] M.E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. *http://dinosaur.compilertools.net/lex/lex.ps*.

[Mat11a] The MathWorks Inc., 3 Apple Hill Drive, Natick, MA 01760-2098. *MATLAB Optimization Toolbox - User's guide*, April 2011.

[Mat11b] The MathWorks Inc., 3 Apple Hill Drive, Natick, MA 01760-2098. *MATLAB Programming Fundamentals*, April 2011.

[MI03] Francesca Mazzia and Felice Iavernero. Test set for initial value problems. Report 40-2003, Department of Mathematics, University of Bari, Via E. Orabano 4, 1-70125, Bari, Italy, 2003. See `http://pitagora.dm.uniba.it/~testset/`.

[MP99] Vijay Menon and Keshav Pingali. A case for source-level transformations in MATLAB. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 53–65, New York, NY, USA, 1999. ACM Press.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, USA, 1997.

[NW99] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer Series in Operations Research. Springer-Verlag, New York, 1999.

[PLW+04] Terence Parr, John Lilly, Peter Wells, Ric Klaren, Mika Illouz, John Mitchell, Scott Stanchfield, Jim Coker, Monty Zukowski, and Chapman Flack. ANTLR Reference Manual. Technical report, MageLang Institute's jGuru.com, January 2004. See ANTLR version 2.7.3 `www.antlr.org/share/1084743321127/ANTLR_Reference_Manual.pdf`.

[PQ95] T. Parr and R. Quong. *Software - Practice and Experience*, volume 25(7), chapter ANTLR: A Predicated LL(k) Parser Generator, pages 789–810. J. Wiley & Sons, July 1995.

[RMG+00] Kenneth H. Rosen, John G. Michaels, Jonathan L. Gross, Jerrold W. Grossman, and Douglas R. Shier. *Handbook of Discrete and Combinatorial Mathematics*. Discrete Mathematics and Its Applications. CRC Press, Boca Raton, Florida, 2000.

[RP99] Luiz De Rose and David Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Trans. Program. Lang. Syst.*, 21(2):286–323, 1999.

[Rum99] S.M. Rump. INTLAB - INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publishers, Dordrecht, 1999. `http://www.ti3.tu-harburg.de/rump/`.

[SGT03] L.F. Shampine, I. Gladwell, and S. Thompson. *Solving ODEs with MATLAB*. Cambridge University Press, 2003.

[SKF05] L. F. Shampine, Robert Ketzscher, and Shaun A. Forth. Using AD to solve BVPs in MATLAB. *ACM Trans. Math. Softw.*, 31(1):79–94, March 2005.

[SR97] L.F. Shampine and M.W. Reichelt. The MATLAB ODE suite. *SIAM J. Sci. Comput.*, 18:1–22, 1997.

[StG11] Richard M. Stallman and the GCC Developer Community. *GNU Compiler Collection Internals*. Free Software Foundation, Boston, MA, March 2011. GCC version 4.6.0 *http://gcc.gnu.org/onlinedocs/gcc-4.6.0/gccint/*.

[Sun] Sun. Java compiler compiler. *https://javacc.dev.java.net*.

[TFK05] Mohamed Tadjouddine, Shaun A. Forth, and Andrew J. Keane. Adjoint differentiation of a structural dynamics solver. In Martin Bücker, George Corliss, Paul Hovland, Uwe Naumann, and Boyana Norris, editors, *AD2004: Proceedings of the 4th International Conference on Automatic Differentiation*, Lecture Notes in Computational Science and Engineering, page To appear. Springer, 2005.

[Veh01] Andre Vehreschild. Semantic augmentation of MATLAB programs to compute derivatives. Master's thesis, Institute for Scientific Computing, Aachen University, Germany, November 2001.

[Wel93] J. B. Wells. Typability and type checking in the second-order lambda-calculus are equivalent and undecidable, 1993.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13:291–299, 1991.

Appendix A

More Problems Comparing Derivative Performances

Table A.1: Ratio CPU($\mathbf{Jf} + \mathbf{f}$)/CPU($\mathbf{f}$) – Jacobian (including function) to function average CPU time ratio for given techniques on the Brusselator problem [SGT03, Kha04, For06] with increasing problem size N

	Ratio CPU($\mathbf{Jf} + \mathbf{f}$)/CPU($\mathbf{f}$)				
n	`numjac`	`msad`	`fmad`	`msad`	`fmad`
$(2 \times N)$	(`comp`)	(`comp`)	(`comp`)	(`sparse`)	(`sparse`)
40	5.21	7.99	92.80	16.12	98.79
80	5.30	7.13	74.43	16.08	83.92
160	5.88	7.12	62.65	16.76	72.05
320	6.46	6.81	48.28	18.10	60.52
640	7.32	6.58	34.39	20.79	50.13
1280	8.12	6.60	23.45	25.25	43.71
2560	8.71	6.75	16.38	33.53	45.00
5120	9.13	7.02	12.63	49.06	56.50
10240	10.25	7.46	11.15	78.80	84.11
20480	11.54	8.62	11.37	138.92	144.32
40960	11.79	8.87	12.50	285.13	295.23

Table A.2: Ratio CPU($\mathbf{Jf} + \mathbf{f}$)/CPU($\mathbf{f}$) – full Jacobian (including function) to function average CPU time ratio on the Burgers' ODE problem [SGT03] with increasing problem size N

n	Ratio CPU($\mathbf{Jf} + \mathbf{f}$)/CPU($\mathbf{f}$)		
	`numjac`	`msad`	`fmad`
$(2 \times N)$	(`full`)	(`full`)	(`full`)
16	19.6	39.5	204.2
32	37.1	37.9	189.0
64	72.9	39.1	170.8
128	145.6	44.2	138.4
256	310.4	90.0	173.5
512	731.1	315.3	491.2
1024	1872.8	724.0	1143.6

Table A.3: Ratio CPU($\mathbf{Jf}+\mathbf{f}$)/CPU($\mathbf{f}$) – sparse Jacobian (including function) to function average CPU time ratio for given techniques on the Burgers' ODE problem [SGT03, Kha04, For06] with increasing problem size N

n	Ratio CPU($\mathbf{Jf} + \mathbf{f}$)/CPU($\mathbf{f}$)				
	`msad`	`fmad`	`numjac`	`msad`	`fmad`
$(2 \times N)$	(`sparse`)	(`sparse`)	(`comp`)	(`comp`)	(`comp`)
16	49.9	219.5	20.4	26.4	205.3
32	50.9	207.0	22.8	25.3	189.0
64	51.8	186.9	23.1	23.2	163.4
128	48.5	144.8	23.3	18.9	117.5
256	50.1	119.1	23.7	16.5	86.1
512	52.6	98.4	24.4	15.1	60.3
1024	55.5	84.2	25.0	14.2	41.2
2048	59.0	77.0	25.7	14.1	30.8
4096	63.9	76.4	26.5	17.4	28.3
8192	73.6	81.7	27.4	20.0	33.2
16384	87.7	94.8	27.7	20.3	32.8
32768	115.6	124.0	27.4	19.7	31.5

Table A.4: Ratio CPU($\nabla\mathbf{f} + \mathbf{f}$)/CPU($\mathbf{f}$) – Gradient (including function) to function average CPU time ratio for given techniques on the Data-fitting problem [BBL+02, For06] with increasing problem size n, with $m = 4$

n	Ratio CPU($\nabla\mathbf{f} + \mathbf{f}$)/CPU($\mathbf{f}$)				
	`numjac` (`full`)	`msad` (`full`)	`fmad` (`full`)	`msad` (`sparse`)	`fmad` (`sparse`)
100	123.04	30.54	107.14	13.88	38.26
200	239.25	71.03	249.30	20.52	39.15
300	349.74	111.32	396.32	27.23	41.36
400	464.17	158.06	553.33	38.67	51.66
500	572.41	205.46	709.77	50.00	62.14
600	654.71	241.18	829.12	56.73	66.67
700	777.54	291.02	1000.00	65.98	76.33
800	870.12	329.88	1134.76	74.55	83.94
900	998.47	385.28	1321.47	85.28	94.48
1000	1120.99	432.72	1486.42	94.75	104.01
1100	1222.19	475.31	1643.12	103.73	112.73
1200	1294.34	505.66	1744.97	109.69	117.19
1300	1317.92	516.67	1798.11	111.32	118.87
1400	1533.23	606.01	2111.71	130.70	139.24
1500	1513.06	593.31	2042.99	126.90	134.18
1600	1721.02	676.11	2327.07	145.51	153.21
1700	1621.79	637.82	2427.24	135.67	141.08
1800	1811.54	714.10	3358.33	153.87	161.61
1900	2034.19	799.35	3787.10	168.27	175.64
2000	2235.58	890.38	4729.49	186.86	194.23

Table A.5: Ratio CPU($\nabla \mathbf{f} + \mathbf{f}$)/CPU($\mathbf{f}$) – Gradient (including function) to function average CPU time ratio for given techniques on the `MINPACK` - dgl1 problem, Homogeneous Superconductors 1-D Ginzburg-Landau [ACMX92, Len05] with increasing problem size n. Entries marked '–' are not available because of memory constraints.

n	Ratio CPU($\nabla \mathbf{f} + \mathbf{f}$)/CPU($\mathbf{f}$)				
	`sfdnls` (`full`)	`msad` (`full`)	`fmad` (`full`)	`msad` (`sparse`)	`fmad` (`sparse`)
8	24.9	21.7	178.0	31.4	176.8
16	25.4	22.4	163.8	31.4	169.7
32	37.1	23.3	156.7	32.0	164.7
64	69.5	29.0	155.3	34.0	158.2
128	134.7	49.4	151.0	34.5	144.6
256	271.7	146.1	268.7	33.8	133.4
512	544.5	446.8	807.3	36.4	187.1
1024	1083.3	1213.5	2086.5	40.9	332.6
2048	–	–	–	44.9	679.1
4096	–	–	–	47.8	1450.6
8192	–	–	–	50.4	–
16384	–	–	–	51.4	–
32768	–	–	–	54.1	–
65536	–	–	–	47.0	–

Table A.6: Ratio CPU($\nabla\mathbf{f} + \mathbf{f}$)/CPU($\mathbf{f}$) – Gradient (including function) to function average CPU time ratio for given techniques on the `MINPACK` - dgl2 problem, Homogeneous Superconductors 2-D Ginzburg-Landau [ACMX92, Len05] with increasing problem size N. Entries marked '–' are not available because of memory constraints.

	Ratio CPU($\nabla\mathbf{f} + \mathbf{f}$)/CPU($\mathbf{f}$)				
n	`sfdnls`	`msad`	`fmad`	`msad`	`fmad`
($4 \times N^2$)	(`full`)	(`full`)	(`full`)	(`sparse`)	(`sparse`)
4	11.2	13.0	80.2	18.6	82.5
16	34.1	12.9	88.7	19.6	93.7
64	114.5	15.5	99.6	21.8	103.8
256	433.4	83.3	205.6	37.0	165.6
1024	1729.8	845.9	1640.8	50.4	189.7
4096	–	–	–	105.7	464.0
16384	–	–	–	253.6	1727.4

Table A.7: Ratio CPU($\nabla\mathbf{f} + \mathbf{f}$)/CPU($\mathbf{f}$) – Gradient (including function) to function average CPU time ratio for given techniques on the `MINPACK` - dssc problem, Steady-State Combustion [ACMX92, Len05] with increasing problem size N. Entries marked '–' are not available because of memory constraints.

	Ratio CPU($\nabla\mathbf{f} + \mathbf{f}$)/CPU($\mathbf{f}$)				
n	`sfdnls`	`msad`	`fmad`	`msad`	`fmad`
(N^2)	(`full`)	(`full`)	(`full`)	(`sparse`)	(`sparse`)
4	6.1	19.0	102.6	26.1	107.7
16	18.9	18.7	110.0	26.0	114.3
64	69.8	18.5	105.8	24.1	107.5
256	271.9	66.0	180.2	22.1	90.7
1024	1073.4	366.5	780.0	22.8	96.1
4096	–	–	–	34.1	223.8
16384	–	–	–	52.3	700.0
65536	–	–	–	77.5	2238.1

Table A.8: Ratio CPU($\mathbf{Jf} + \mathbf{f}$)/CPU($\mathbf{f}$) – Jacobian (including function) to function average CPU time ratio for given techniques on ODE problems from [SGT03, Kha04] and optimisation problems from [ACMX92, Len05] of size n

		Ratio CPU($\mathbf{Jf} + \mathbf{f}$)/CPU($\mathbf{f}$)		
Problem	n	sfdnls	`msad`	`fmad`
Coating Thickness Standardization	134	256.28	49.65	107.87
Pollution ODE	20	10.86	9.84	113.37
Combustion of Propane - Full	11	22.22	35.03	394.29
Human Heart Dipole	8	23.12	53.08	737.16
Chemical AzkoNobel	6	16.24	17.22	252.71
Combustion of Propane - Reduced	5	20.67	64.94	921.80
Amplifier DAE	5	13.86	15.08	170.11
Enzyme Reaction	4	18.69	9.51	111.89
Robertson ODE	3	11.05	10.48	124.22

Appendix B

Function Run Data

Table B.1: Problem type, size (m, n), maximum entries in Jacobian row $\hat{n}$, number of directional derivatives used in compression p and function CPU times of large scale problems from MATLAB Optimisation Toolbox

Problem	Problem type	(m, n)	$\hat{n}$	p	CPU($\mathbf{f}$) (ms)
`nlsf1a`	Function Jacobian	(1000,1000)	3	3	0.363
`brownf`	gradient from function	(1,1000)	1000	1000	0.668
`browng`	Hessian from gradient	(1000,1000)	3	3	2.393
`tbroyf`	gradient from function	(1,800)	800	800	1.272
`tbroyg`	Hessian from gradient	(800,800)	6	8	3.558

Table B.2: Function CPU times for the Brusselator problem

Problem size n	40	80	160	320	640	1280	2560	5120	10240
Function CPU(ms)	0.193	0.239	0.318	0.474	0.794	1.430	2.701	5.277	10.446

Table B.3: Function CPU times for the Burgers' ODE problem

Problem size n	16	32	64	128	256	512	1024
Function CPU(ms)	0.515	0.506	0.502	0.494	0.616	0.875	1.394

Table B.4: Function CPU times for the Burgers' ODE problem (non-vectorised)

Problem size n	16	32	64	128	256	512	1024
Function CPU(ms)	0.0745	0.0818	0.0964	0.1248	0.1826	0.2988	0.5452

Table B.5: Absolute run-times for complete solutions to boundary value problems using `bvp4c` and `bvp4cAD`

Problem	CPU time (s)		
	`bvp4c`	`bvp4cAD` `(msad)`	`bvp4cAD` `(fmad)`
`I`	0.05	0.03	0.07
`II`	0.14	0.06	0.45
`III`	0.24	0.15	1.22
`IV`	0.47	0.24	1.74
`V`	0.50	0.20	1.28
`VI`	1.29	0.44	1.27
`VII`	0.56	0.78	5.37
`VIII`	0.05	0.07	0.43

Appendix C

Complete Examples of Augmented AD output

C.1 MINPACK-2 Enzyme reaction problem

C.1.1 Input code

```
function f = fdaer(x)
%! active(x), size(x) = [4, 1]

  v = [ 4.0d0;  2.0d0;  1.0d0;  5.0d-1;  2.5d-1; 1.67d-1;...
        1.25d-1;  1.0d-1;  8.33d-2; 7.14d-2;  6.25d-2 ];
  y = [ 1.957d-1;  1.947d-1;  1.735d-1;  1.6d-1; 8.44d-2;...
        6.27d-2; 4.56d-2;  3.42d-2; 3.23d-2;  2.35d-2;  2.46d-2 ];

  f = y - x(1).*(v.*(v+x(2)))./(v.*(v+x(3)) + x(4));
```

C.1.2 Augmented AD code

```
function [f_0, ad_f_1] = ad_fdaer(x_0, ad_x_0)

  n_derivs_8 = floor(numel(ad_x_0) ./ numel(x_0));
  ad_x_1 = reshape(ad_x_0, [numel(x_0),n_derivs_8]);

  v_0 = [ 4.0d0;  2.0d0;  1.0d0;  5.0d-1;  2.5d-1; 1.67d-1;...
```

```
7           1.25d-1;  1.0d-1;  8.33d-2; 7.14d-2;  6.25d-2 ];
8   y_0 = [ 1.957d-1;  1.947d-1;  1.735d-1;  1.6d-1; 8.44d-2;...
9           6.27d-2; 4.56d-2;  3.42d-2; 3.23d-2;  2.35d-2;  2.46d-2 ];
10
11  op_arr_1 = x_0(1);
12
13  d_op_arr_4 = ad_x_1(1, :);
14  op_arr_3 = x_0(2);
15
16  d_op_arr_9 = ad_x_1(2, :);
17  z_0 = v_0 + op_arr_3;
18
19  d_z_0 = d_op_arr_9(ones(11,1), :);
20  ad_MTmp3_0 = d_z_0;
21  z_3 = v_0 .* (z_0);
22
23  n_derivs_0 = floor(numel(ad_MTmp3_0) ./ 11);
24
25  tmp_mults__0 = v_0(1:11);
26  d_z_17 = tmp_mults__0(1:11, ones(1,n_derivs_0)) .* ad_MTmp3_0;
27  MTmp5_0 = (z_3);
28  ad_MTmp5_0 = d_z_17;
29  z_5 = op_arr_1 .* MTmp5_0;
30
31  d_prod1_8 = op_arr_1 .* ad_MTmp5_0;
32  tmp_mults__2 = MTmp5_0(1:11);
33  d_prod2_15 = tmp_mults__2 * d_op_arr_4;
34  d_z_19 = d_prod1_8 + d_prod2_15;
35
36  op_arr_5 = x_0(3);
37
38  d_op_arr_14 = ad_x_1(3, :);
39  z_6 = v_0 + op_arr_5;
40
41  d_z_20 = d_op_arr_14(ones(11,1), :);
42  ad_MTmp9_0 = d_z_20;
43  z_9 = v_0 .* (z_6);
44
45  n_derivs_4 = floor(numel(ad_MTmp9_0) ./ 11);
```

```
46
47  tmp_mults__4 = v_0(1:11);
48  d_z_37 = tmp_mults__4(1:11, ones(1,n_derivs_4)) .* ad_MTmp9_0;
49  op_arr_7 = x_0(4);
50
51  d_op_arr_19 = ad_x_1(4, :);
52  z_11 = z_9 + op_arr_7;
53
54  d_z_53 = d_z_37 + d_op_arr_19(ones(1,11), :);
55  ad_MTmp13_0 = d_z_53;
56  inv_y_0 = 1 ./ (z_11);
57  z_13 = z_5 .* inv_y_0;
58
59  n_derivs_6 = floor(numel(d_z_19) ./ 11);
60
61  tmp_mults__6 = z_13(1:11);
62  d_prod1_27 = tmp_mults__6(1:11, ones(1,n_derivs_6)) .* ad_MTmp13_0;
63  d_sum1_0 = d_z_19 - d_prod1_27;
64  tmp_mults__7 = inv_y_0(1:11);
65  d_z_67 = tmp_mults__7(1:11, ones(1,n_derivs_6)) .* d_sum1_0;
66  f_0 = y_0 - z_13;
67
68  ad_f_0 = -d_z_67;
69  ad_f_1 = reshape(ad_f_0, [numel(f_0),n_derivs_8]);
```

C.2 MATLAB Brown's minimisation problem

C.2.1 Input code

```
function f = fbrown(x)
%BROWNFG Nonlinear minimization test problem

%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.6.4.2 $  $Date: 2004/04/01 16:13:02 $
%   Thomas F. Coleman 7-1-96

%! active(x), sparse_der(x)

% Evaluate the function.

  n = length(x);

  i = 1:(n-1);
  f = sum((x(i).^2).^(x(i+1).^2+1) + (x(i+1).^2).^(x(i).^2+1));
```

C.2.2 Augmented AD code

```
function [f_0, ad_f_0] = ad_fbrown(x_0, ad_x_0)
  %BROWNFG Nonlinear minimization test problem

  %   Copyright 1990-2004 The MathWorks, Inc.
  %   $Revision: 1.6.4.2 $  $Date: 2004/04/01 16:13:02 $
  %   Thomas F. Coleman 7-1-96


  n_derivs_12 = floor(numel(ad_x_0) ./ numel(x_0));
  ad_x_1 = reshape(ad_x_0, [numel(x_0),n_derivs_12]);

  n_0 = length(x_0);

  i_0 = 1:(n_0 - 1);
  op_arr_0 = x_0(i_0);

```

```
17  if isscalar(i_0)
18    d_op_arr_0 = ad_x_1(i_0, :);
19  else
20    tmp_indx__0 = reshape((1:numel(x_0)), size(x_0));
21    tmp_indx__1 = tmp_indx__0(i_0);
22    d_op_arr_0 = ad_x_1(tmp_indx__1(:), :);
23  end
24  z_1 = op_arr_0 .* op_arr_0;
25
26  tmp_ssb__0 = numel(op_arr_0);
27  n_derivs_0 = floor(numel(d_op_arr_0) ./ tmp_ssb__0);
28
29  tmp_mults__0 = 2 .* op_arr_0;
30  tmp_mults__1 = tmp_mults__0(:);
31  tmp_ssa__0 = numel(tmp_mults__1);
32  if tmp_ssa__0 == tmp_ssb__0
33    d_z_1 = tmp_mults__1(:, ones(1,n_derivs_0)) .* d_op_arr_0;
34  elseif tmp_ssb__0 == 1
35    d_z_1 = tmp_mults__1 * d_op_arr_0;
36  elseif tmp_ssa__0 == 1
37    d_z_1 = tmp_mults__1 .* d_op_arr_0;
38  end
39
40  MTmp4_0 = (z_1);
41  MTmp5_0 = i_0 + 1;
42  op_arr_2 = x_0(MTmp5_0);
43
44  if isscalar(MTmp5_0)
45    d_op_arr_5 = ad_x_1(MTmp5_0, :);
46  else
47    tmp_indx__2 = reshape((1:numel(x_0)), size(x_0));
48    tmp_indx__3 = tmp_indx__2(MTmp5_0);
49    d_op_arr_5 = ad_x_1(tmp_indx__3(:), :);
50  end
51  z_3 = op_arr_2 .* op_arr_2;
52
53  tmp_ssb__2 = numel(op_arr_2);
54  n_derivs_2 = floor(numel(d_op_arr_5) ./ tmp_ssb__2);
55
```

```
  tmp_mults__4 = 2 .* op_arr_2;
  tmp_mults__5 = tmp_mults__4(:);
  tmp_ssa__2 = numel(tmp_mults__5);
  if tmp_ssa__2 == tmp_ssb__2
    d_z_19 = tmp_mults__5(:, ones(1,n_derivs_2)).* d_op_arr_5;
  elseif tmp_ssb__2 == 1
    d_z_19 = tmp_mults__5 * d_op_arr_5;
  elseif tmp_ssa__2 == 1
    d_z_19 = tmp_mults__5 .* d_op_arr_5;
  end

  z_5 = z_3 + 1;

  MTmp9_0 = (z_5);
  ad_MTmp9_0 = d_z_19;
  tmp_z_2 = MTmp4_0 .^ (MTmp9_0 - 1);
  z_6 = tmp_z_2 .* MTmp4_0;

  tmp_ssb__4 = numel(MTmp4_0);
  n_derivs_4 = floor(numel(d_z_1) ./ tmp_ssb__4);

  tmp_mults__8 = MTmp9_0 .* tmp_z_2;
  tmp_mults__9 = tmp_mults__8(:);
  tmp_ssa__4 = numel(tmp_mults__9);
  if tmp_ssa__4 == tmp_ssb__4
    d_prod1_23 = tmp_mults__9(:, ones(1,n_derivs_4)) .* d_z_1;
  elseif tmp_ssb__4 == 1
    d_prod1_23 = tmp_mults__9 * d_z_1;
  elseif tmp_ssa__4 == 1
    d_prod1_23 = tmp_mults__9 .* d_z_1;
  end

  tmp_ssb__5 = numel(MTmp9_0);
  n_derivs_5 = floor(numel(ad_MTmp9_0) ./ tmp_ssb__5);

  tmp_mults__10 = z_6 .* log(MTmp4_0);
  tmp_mults__11 = tmp_mults__10(:);
  tmp_ssa__5 = numel(tmp_mults__11);
  if tmp_ssa__5 == tmp_ssb__5
```

```
95      d_prod2_23 = tmp_mults__11(:, ones(1,n_derivs_5)) .* ad_MTmp9_0;
96    elseif tmp_ssb__5 == 1
97      d_prod2_23 = tmp_mults__11 * ad_MTmp9_0;
98    elseif tmp_ssa__5 == 1
99      d_prod2_23 = tmp_mults__11 .* ad_MTmp9_0;
100   end
101
102   d_z_20 = d_prod1_23 + d_prod2_23;
103   MTmp11_0 = i_0 + 1;
104   op_arr_4 = x_0(MTmp11_0);
105
106   if isscalar(MTmp11_0)
107     d_op_arr_10 = ad_x_1(MTmp11_0, :);
108   else
109     tmp_indx__4 = reshape((1:numel(x_0)), size(x_0));
110     tmp_indx__5 = tmp_indx__4(MTmp11_0);
111     d_op_arr_10 = ad_x_1(tmp_indx__5(:), :);
112   end
113   z_9 = op_arr_4 .* op_arr_4;
114
115   tmp_ssb__7 = numel(op_arr_4);
116   n_derivs_7 = floor(numel(d_op_arr_10) ./ tmp_ssb__7);
117
118   tmp_mults__14 = 2 .* op_arr_4;
119   tmp_mults__15 = tmp_mults__14(:);
120   tmp_ssa__7 = numel(tmp_mults__15);
121   if tmp_ssa__7 == tmp_ssb__7
122     d_z_23 = tmp_mults__15(:, ones(1,n_derivs_7)) .* d_op_arr_10;
123   elseif tmp_ssb__7 == 1
124     d_z_23 = tmp_mults__15 * d_op_arr_10;
125   elseif tmp_ssa__7 == 1
126     d_z_23 = tmp_mults__15 .* d_op_arr_10;
127   end
128
129   MTmp14_0 = (z_9);
130   op_arr_7 = x_0(i_0);
131
132   if isscalar(i_0)
133     d_op_arr_19 = ad_x_1(i_0, :);
```

```
134   else
135     tmp_indx__6 = reshape((1:numel(x_0)), size(x_0));
136     tmp_indx__7 = tmp_indx__6(i_0);
137     d_op_arr_19 = ad_x_1(tmp_indx__7(:), :);
138   end
139   z_10 = op_arr_7 .* op_arr_7;
140
141   tmp_ssb__9 = numel(op_arr_7);
142   n_derivs_9 = floor(numel(d_op_arr_19) ./ tmp_ssb__9);
143
144   tmp_mults__18 = 2 .* op_arr_7;
145   tmp_mults__19 = tmp_mults__18(:);
146   tmp_ssa__9 = numel(tmp_mults__19);
147   if tmp_ssa__9 == tmp_ssb__9
148     d_z_41 = tmp_mults__19(:, ones(1,n_derivs_9)) .* d_op_arr_19;
149   elseif tmp_ssb__9 == 1
150     d_z_41 = tmp_mults__19 * d_op_arr_19;
151   elseif tmp_ssa__9 == 1
152     d_z_41 = tmp_mults__19 .* d_op_arr_19;
153   end
154
155   z_13 = z_10 + 1;
156
157   MTmp18_0 = (z_13);
158   ad_MTmp18_0 = d_z_41;
159   tmp_z_5 = MTmp14_0 .^ (MTmp18_0 - 1);
160   z_15 = tmp_z_5 .* MTmp14_0;
161
162   tmp_ssb__10 = numel(MTmp14_0);
163   n_derivs_10 = floor(numel(d_z_23) ./ tmp_ssb__10);
164
165   tmp_mults__20 = MTmp18_0 .* tmp_z_5;
166   tmp_mults__21 = tmp_mults__20(:);
167   tmp_ssa__10 = numel(tmp_mults__21);
168   if tmp_ssa__10 == tmp_ssb__10
169
170     d_prod1_47 = tmp_mults__21(:, ones(1,n_derivs_10)) .* d_z_23;
171   elseif tmp_ssb__10 == 1
172     d_prod1_47 = tmp_mults__21 * d_z_23;
```

```
173   elseif tmp_ssa__10 == 1
174     d_prod1_47 = tmp_mults__21 .* d_z_23;
175   end
176
177   tmp_ssb__11 = numel(MTmp18_0);
178   n_derivs_11 = floor(numel(ad_MTmp18_0) ./ tmp_ssb__11);
179
180   tmp_mults__22 = z_15 .* log(MTmp14_0);
181   tmp_mults__23 = tmp_mults__22(:);
182   tmp_ssa__11 = numel(tmp_mults__23);
183   if tmp_ssa__11 == tmp_ssb__11
184     d_prod2_47 = tmp_mults__23(:, ones(1,n_derivs_11)) .* ad_MTmp18_0;
185   elseif tmp_ssb__11 == 1
186     d_prod2_47 = tmp_mults__23 * ad_MTmp18_0;
187   elseif tmp_ssa__11 == 1
188     d_prod2_47 = tmp_mults__23 .* ad_MTmp18_0;
189   end
190
191   d_z_43 = d_prod1_47 + d_prod2_47;
192   z_17 = z_6 + z_15;
193
194   ssx_2 = numel(z_6);
195   ssy_2 = numel(z_15);
196   if ssx_2 == ssy_2
197     d_z_59 = d_z_20 + d_z_43;
198   elseif ssx_2 == 1
199     d_z_59 = d_z_20(ones(1,ssy_2), :) + d_z_43;
200   elseif ssy_2 == 1
201     d_z_59 = d_z_20 + d_z_43(ones(1,ssx_2), :);
202   else
203     error('internal error in plus');
204   end
205
206   [f_0, ad_f_0] = ad_sum(z_17, d_z_59);
207   ad_f_0 = reshape(ad_f_0, [numel(f_0),n_derivs_12]);
```

Appendix D

Debug output from MSAD

D.1 SCCP lattice inference pass output for program in Figure 3.34

```
807. 94   ( () isscalar ( array_indx_list_ x ) )      /* Operation */
MTmp0_0                                               /* LHS variable */
00000000000000000001000000000000  LOGICAL             /* Inferred class */
rank: 0000000000000011  MSCALAR                       /* Inferred rank */
complex: FALSE  sparse: FALSE  const_value: UP_BOUND  /* Other attrs. */
size: [ 1 1 ]                                         /* Inferred size */

6. 47   ( ~ MTmp0 )
y_0
00000000000000000001000000000000  LOGICAL
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: UP_BOUND
size: [ 1 1 ]

7. 37   ( / 3 2 )
t_0
00000000000000000000001000000000  DOUBLE
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: TRUE
size: [ 1 1 ]
{ 1.5 }                                               /* Inferred value */
```

```
815. 41   ( <= y 0 )
MVar0_0
00000000000000000001000000000000  LOGICAL
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: UP_BOUND
size: [ 1 1 ]

811. 47   ( ~ MVar0 )
MTmp1_0
00000000000000000001000000000000  LOGICAL
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: UP_BOUND
size: [ 1 1 ]

817. 94   ( () pi array_indx_list )
MTmp2_0
00000000000000000000001000000000  DOUBLE
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: TRUE
size: [ 1 1 ]
{ 3.14159 }

818. 37   ( / 1 t )
MTmp3_0
00000000000000000000001000000000  DOUBLE
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: TRUE
size: [ 1 1 ]
{ 0.666667 }

819. 81   (  () MTmp3 )
MTmp4_0
00000000000000000000001000000000  DOUBLE
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: TRUE
size: [ 1 1 ]
{ 0.666667 }
```

```
9. 34   ( * MTmp2 MTmp4 )
z_2
00000000000000000000001000000000  DOUBLE
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: TRUE
size: [ 1 1 ]
{ 2.0944 }

11. 94   ( () ones ( array_indx_list_ 2 ) )
t_1
00000000000000000000001000000000  DOUBLE
rank: 0000000000001111  MMATRIX
complex: FALSE  sparse: FALSE  const_value: TRUE
size: [ 2 2 ]
{ 1 1 1 1 }

12. 34   ( * x t )
z_1
00000000000000000000000111111110  INTEGER
rank: 0000000000001111  MMATRIX
complex: UP_BOUND  sparse: FALSE  const_value: UP_BOUND
size: [ Inv 2 ]

811. 47   ( ~ MVar0 )

825. 94   ( () phi ( array_indx_list_ z z ) )
z_3
00000000000000000000001000000000  DOUBLE
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: TRUE
size: [ 1 1 ]
{ 2.0944 }

825. 94   ( () phi ( array_indx_list_ z z ) )
z_3
00000000000000000000001111111110  NUMERIC
rank: 0000000000001111  MMATRIX
complex: UP_BOUND  sparse: FALSE  const_value: UP_BOUND
size: [ Inv Inv ]
```

```
9. 34   ( * MTmp2 MTmp4 )

826. 94   ( () phi ( array_indx_list_ z null_ssa ) )
z_0
00000000000000000000001111111110  NUMERIC
rank: 0000000000001111  MMATRIX
complex: UP_BOUND  sparse: FALSE  const_value: UP_BOUND
size: [ Inv Inv ]

819. 81   (  () MTmp3 )

9. 34   ( * MTmp2 MTmp4 )

12. 34   ( * x t )
```

SCCP lattice inference output for program in Figure 3.34

```
807. 94   ( () isscalar ( array_indx_list_ x ) )      /* Operation */
MTmp0_0                                               /* LHS variable */
00000000000000000001000000000000  LOGICAL            /* Inferred class */
rank: 0000000000000011  MSCALAR                       /* Inferred rank */
complex: FALSE  sparse: FALSE  const_value: TRUE      /* Other attrs. */
size: [ 1 1 ]                                         /* Inferred size */
{ 1 }                                                 /* Inferred value */

6. 47   ( ~ MTmp0 )
y_0
00000000000000000001000000000000  LOGICAL
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: TRUE
size: [ 1 1 ]
{ 0 }

7. 37   ( / 3 2 )
t_0
00000000000000000000001000000000  DOUBLE
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: TRUE
size: [ 1 1 ]
```

```
{ 1.5 }

815. 41   ( <= y 0 )
MVar0_0
00000000000000000001000000000000  LOGICAL
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: TRUE
size: [ 1 1 ]
{ 1 }

811. 47   ( ~ MVar0 )
MTmp1_0
00000000000000000001000000000000  LOGICAL
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: TRUE
size: [ 1 1 ]
{ 0 }

817. 94   ( () pi array_indx_list )
MTmp2_0
00000000000000000000001000000000  DOUBLE
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: TRUE
size: [ 1 1 ]
{ 3.14159 }

818. 37   ( / 1 t )
MTmp3_0
00000000000000000000001000000000  DOUBLE
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: TRUE
size: [ 1 1 ]
{ 0.666667 }

819. 81   (  () MTmp3 )
MTmp4_0
00000000000000000000001000000000  DOUBLE
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: TRUE
```

```
size: [ 1 1 ]
{ 0.666667 }

9. 34   ( * MTmp2 MTmp4 )
z_2
00000000000000000000001000000000  DOUBLE
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: TRUE
size: [ 1 1 ]
{ 2.0944 }

825. 94   ( () phi ( array_indx_list_ z z ) )
z_3
00000000000000000000001000000000  DOUBLE
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: TRUE
size: [ 1 1 ]
{ 2.0944 }

811. 47   ( ~ MVar0 )

826. 94   ( () phi ( array_indx_list_ z null_ssa ) )
z_0
00000000000000000000001000000000  DOUBLE
rank: 0000000000000011  MSCALAR
complex: FALSE  sparse: FALSE  const_value: TRUE
size: [ 1 1 ]
{ 2.0944 }

9. 34   ( * MTmp2 MTmp4 )
819. 81   (  () MTmp3 )
9. 34   ( * MTmp2 MTmp4 )
```

SCCP lattice inference output after modifying program in Figure 3.34 by adding MSAD directive: `%!  size(x) = [1 1]`

D.2 IR code after specialisation of overloaded `fmad - plus` operation in Figure 3.40

```
1  function [z_3, d_z_17] = plus(x_2, d_x_1, y_5, d_y_1)
2
3    ENTRY_BLOCK_START 'MLbl12'
4    ENTRY_BLOCK_END 'MLbl12'
5
6    BLOCK_START 'MLbl0'
7      z_2 = x_2 + y_5;
8    BLOCK_END 'MLbl0'
9
10   % if
11     BLOCK_START 'MLbl15'
12     BLOCK_END 'MLbl15'
13
14     BLOCK_START 'MLbl1'
15       ssx_0 = numel(x_2);
16       ssy_0 = numel(y_5);
17     BLOCK_END 'MLbl1'
18     % if
19       BLOCK_START 'MLbl17'
20         MVar1_0 = ssx_0 == ssy_0;
21         MTmp3_0 = ~MVar1_0;
22         goto MTmp3_0 ? MLbl18;
23       BLOCK_END 'MLbl17'
24
25       BLOCK_START 'MLbl2'
26         d_z_2 = d_x_1 + d_y_1;
27         goto MLbl16;
28       BLOCK_END 'MLbl2'
29     % elseif
30       BLOCK_START 'MLbl18'
31         MVar2_0 = ssx_0 == 1;
32         MTmp4_0 = ~MVar2_0;
33         goto MTmp4_0 ? MLbl21;
34       BLOCK_END 'MLbl18'
35       % if
36         BLOCK_START 'MLbl20'
```

```
37            goto MLbl4
38          BLOCK_END 'MLbl20'
39        % else
40          BLOCK_START 'MLbl4'
41            MTmp19_0 = ones(1, ssy_0);
42            MTmp20_0 = m_builtin_end(d_x_1, 2, 2);
43            MTmp21_0 = 1:MTmp20_0;
44            MTmp22_0 = Array_access(d_x_1,{MTmp19_0,MTmp21_0});
45            d_z_8 = MTmp22_0 + d_y_1;
46          BLOCK_END 'MLbl4'
47
48          BLOCK_START 'MLbl19'
49            d_z_9 = phi(d_z_8, d_z_7);
50            goto MLbl16;
51          BLOCK_END 'MLbl19'
52        % end if
53      % elseif
54        BLOCK_START 'MLbl21'
55          MVar4_0 = ssy_0 == 1;
56          MTmp23_0 = ~MVar4_0;
57          goto MTmp23_0 ? MLbl7;
58        BLOCK_END 'MLbl21'
59        % if
60          BLOCK_START 'MLbl23'
61            goto MLbl6
62          BLOCK_END 'MLbl23'
63        % else
64          BLOCK_START 'MLbl6'
65            MTmp38_0 = ones(1, ssx_0);
66            MTmp39_0 = m_builtin_end(d_y_1, 2, 2);
67            MTmp40_0 = 1:MTmp39_0;
68            MTmp41_0 = Array_access(d_y_1,{MTmp38_0,MTmp40_0});
69            d_z_6 = d_x_1 + MTmp41_0;
70          BLOCK_END 'MLbl6'
71
72          BLOCK_START 'MLbl22'
73            d_z_4 = phi(d_z_6, d_z_5);
74            goto MLbl16;
75          BLOCK_END 'MLbl22'
```

```
76        % end if
77      % else
78        BLOCK_START 'MLbl7'
79          error('internal error in plus');
80        BLOCK_END 'MLbl7'
81
82        BLOCK_START 'MLbl16'
83          d_z_3 = phi(d_z_9, d_z_2, d_z_4, null_ssa);
84          goto MLbl14;
85        BLOCK_END 'MLbl16'
86      % end if
87    % elseif
88      % if
89      % else
90      % end if
91    % elseif
92      % if
93      % else
94      % end if
95      BLOCK_START 'MLbl14'
96        d_z_10 = phi(null_ssa, d_z_16, d_z_11, d_z_3);
97      BLOCK_END 'MLbl14'
98    % end if
99
100   EXIT_BLOCK_START 'MLbl13'
101     z_3 = phi(z_2, null_ssa);
102     d_z_17 = phi(d_z_10, null_ssa);
103   EXIT_BLOCK_END 'MLbl13'
```

Appendix E

Invoking MSAD and accompanying tests

Chapter 4 and Appendix A presented results using MSAD on an range of optimisation problems from MATLAB [Mat11a] and MINPACK [AM91, Len05] test sets, ODE problems from the MATLAB ODE suite [SGT03] and the CWI IVP problem set [MI03], and BVP problems from Shampine, Ketzscher and Forth [SKF05]. A large portion of the test harness is inherited from MAD and updated to be used with MSAD. The software disk accompanying this thesis includes all the necessary material to re-run the tests using MSAD. This chapter describes the necessary top-level scripts and commands required to run these tests.

E.1 Setup

The accompanying software disk includes the following items:

1. `MSAD`
2. `MAD`
3. `BVP_tests`
4. `ODE_tests`
5. `OPTIM_tests`
6. `MSAD_sources`

MSAD is supplied in run-time form and in source. The pre-built binaries are availble in the `MSAD/bin` directory and only supplied for Windows XP/Vista/7 hosts based on MinGW (`http://www.mingw.org/`) and Cygwin (`http://www.cygwin.com/`) run-time. Binaries for UNIX/Linux will need to be built from the MSAD sources. The following Windows command line executables are included:

1. `MSAD.exe - 32-bit MinGW (dos seperator)`
2. `MSAD_i686_32_cygwin_unix.exe - 32-bit Cygwin (unix separator)`
3. `MSAD_i686_32_mingw_dos.exe - 32-bit MinGW (dos seperator)`
4. `MSAD_i686_32_mingw_unix.exe - 32-bit MinGW (unix seperator)`
5. `MSAD_x86_64_mingw_dos.exe - 64-bit MinGW (dos seperator)`

We recommend the default `MSAD.exe` be used, unless a particular flavour is required. The UNIX/Linux variants can be built by compiling the MSAD sources using the following sequence of commands in a shell:

```
1    cd $MSAD_TESTS/MSAD_sources/antlr-2.7.7
2    make clean
3    ./configure --with-cxxflags="-O2 -std=c++0x -fno-strict-aliasing -g"
4    make
5    cd $MSAD_TESTS/MSAD_sources
6    make clean
7    make BUILD=unix HOST=unix
```

For the sake of completeness and reproducibility the MAD package [For06] used for testing has also been included above. The test harnesses use both MAD and MSAD and requires the two packages to be recognised by MATLAB. For convenience we assume the contents of the disk are copied to a location `$MSAD_TESTS`, which is accessible both by MATLAB and a shell, Windows or Unix/Linux. The following sequence of MATLAB commands will register both packages in MATLAB:

```
1    global MAD
2    MAD = '$MSAD_TESTS/MAD'
3    cd $MSAD_TESTS/MAD
4    startupmad
5    path(path, '$MSAD_TESTS/MSAD/scratch');
```

```
6     path(path, '$MSAD_TESTS/MSAD/globals');
7     global MSAD
8     MSAD='$MSAD_TESTS/MSAD'
```

Before proceeding with the following tests, it may be useful to ensure the MATLAB path environment is set-up correctly by executing the MATLAB `'path'` command. The trailing few entries should contain the following paths (the leading default entries, skipped here, will vary depending on the MATLAB version and other toolboxes installed):

```
$MSAD_TESTS\MAD\madutil
$MSAD_TESTS\MAD\madrecode
$MSAD_TESTS\MAD\madtapeutil
$MSAD_TESTS\MAD\madadjfuncs
$MSAD_TESTS\MAD\OptimToolbox
$MSAD_TESTS\MAD
$MSAD_TESTS\MAD\MADEXAMPLES
$MSAD_TESTS\MAD\MADEXAMPLES\MADbasic
$MSAD_TESTS\MAD\MADEXAMPLES\MADBlackBox
$MSAD_TESTS\MAD\MADEXAMPLES\MADimplicit_eqs
$MSAD_TESTS\MAD\MAD\MADEXAMPLES\MADodes
$MSAD_TESTS\MAD\MAD\MADEXAMPLES\MADoptimization
$MSAD_TESTS\MAD\MAD\MADEXAMPLES\MADMinpack
$MSAD_TESTS\MAD\MSAD\scratch
$MSAD_TESTS\MAD\MSAD\globals
```

E.2 Running BVP tests

The BVP tests described in Section 4.3 can be run from MATLAB by changing to `$MSAD_TESTS/BVP_tests` by invoking the `timeall.m` file. The `timeall` function also accepts the number of iterations to run as an optional argument. In case of the BVP tests, the included `bvp4cAD` solver invokes MSAD internally to generate the required AD files as necessary. If the problem files are subsequently modified, the AD files will be re-generated based on time-stamps.

E.3 Running ODE tests

The ODE tests described in Section 4.1 can be run from MATLAB by changing to `$MSAD_TESTS/ODE_tests`. There are two tests included in this case, determining the solution to Burgers' ODE problem and the Brusselator ODE problem from MATLAB. Each test has an accompanying test script, `testBurgersodeSolve.m` in sub-directory `burger`, and `testBrussodeSolve.m` in sub-directory `bruss` to solve the Burgers' ODE problem and the Brusselator ODE problem respectively. The associated AD files providing derivatives are not automatically generated by the test harness in this case, but are pre-generated and placed along with the remaining files. In case they need to be re-generated, the following single command may be used as a template to invoke MSAD from the command line to generate AD file for the Burgers' ODE problem to compute derivatives using compressed or full mode (unix shell assumed here, change path separator for windows):

```
1  bin/MSAD.exe -base-file=tests/fBurgersode.m
     -out-file=scratch/ad_fBurgersodeNonVec.m -bi-ad-dir=globals/
     -default-inactive -default-full -use-full-ders -fw-ad
```

Similarly for computing derivatives using sparse mode use the following single line command:

```
1  bin/MSAD.exe -base-file=tests/fBurgersode.m
     -out-file=scratch/ad_sfBurgersodeNonVec.m -bi-ad-dir=globals/
     -default-inactive -default-full -use-sparse-ders -fw-ad
```

E.4 Running Optimisation tests

The Optimisation tests described in Section 4.2 can be run from MATLAB by changing to `$MSAD_TESTS/OPTIM_tests`. There are nine tests included in this directory. Each test has a `batch*.m` script included which runs the solver across several problem sizes, supplying derivatives using different methods (finite-differencing, MAD compressed, MAD sparse, MSAD compressed,

MSAD sparse and hand-cded), and using `large` or `medium` type optimisation solvers. The problems and associated top-level test files are listed below:

1. `brown` - Brown's nonlinear minimisation problem from MATLAB solved using MATLAB unconstrained minimisation solver `fminunc`, run `batchTestBrown.m`

2. `cpf_lsq` - Propane combustion problem from MINPACK-2 solved using MATLAB least-squares method `lsqnonlin`, run `batchTestDcpf.m`

3. `cpf_nonlin` - Propane combustion problem from MINPACK-2 solved using MATLAB non-linear equation solver `fsolve`, run `batchTestDcpf.m`

4. `cts` - Coating Thickness Standardization problem from MINPACK-2 solved using MATLAB least-squares method `lsqnonlin`, run `batchTestDcts.m`

5. `gl2` - Ginzburg-Landau superconductivity problem from MINPACK-2 solved using MATLAB unconstrained minimisation solver `fminunc`, run `batchTestDgl2.m`

6. `hhd_nonlin` - Human Heart Dipole problem from MINPACK-2 solved using MATLAB nonlinear equation solver `fsolve`, run `batchTestDhhd`

7. `nlsf1a` - NSLF1A nonlinear vector problem from MATLAB solved using MATLAB nonlinear equation solver `fsolve`, run `batchTestNlsf1a.m`

8. `ssc` - Steady State Combustion problem from MINPACK-2 solved using MATLAB unconstrained minimisation solver `fminunc`, run `batchTestDssc.m`

9. `tbroy` - TBROY problem from MATLAB solved using MATLAB solved using MATLAB constrained minimisation solver `fmincon`, run `batchTestTbroy.m`

Similar to the ODE problems, the AD files providing derivatives are not automatically generated by the test harness, but are pre-generated and placed

with the remaining files. If they need to be re-generated the MSAD commands similar to those in the previous Section E.3 can be used to do so.

E.5 Comparing derivative performance

The tests in the previous Sections E.3 and E.4 determine complete numerical solutions by supplying derivatives generated using MSAD, and compare the solution and absolute run-time with other methods providing derivatives to the respective solvers. This section lists the top-level scripts required to gather performance statistics of computing the derivatives, CPU($\mathbf{Jf} + \mathbf{f}$) and CPU($\mathbf{Jf} + \mathbf{f}$)/CPU($\mathbf{f}$) alone, as tabulated in Section 4.1 and Section 4.2.

The tests related to computing derivative performance are placed in the MSAD test folder itself, `$MSAD_TESTS/MSAD/tests`. Each of the problems have a corresponding `batch*.m` file to compute and time the derivative calculations over a range of problem sizes. The following tests are available:

1. `batchBrussode.m` - Brusselator ODE problem
2. `batchBurgersode.m` - Burgers' ODE problem
3. `batchBurgersodeNonVec.m` - Burgers' ODE problem (non-vectorised)
4. `batchDataFit.m` - Data-fitting problem
5. `batchDgl1.m` - Ginzburg-Landau 1-D problem
6. `batchDgl2.m` - Ginzburg-Landau 2-D problem
7. `batchDsfi.m` - Solid fuel ignition problem
8. `batchDssc.m` - Steady State Combustion problem

E.6 MSAD command line arguments

The complete MSAD command line syntax is shown below (generated by running `bin/MSAD.exe --help`):

```
Syntax:
  MSAD.exe -base-file=<input M-file>
           [ -out-file=<output M-file> ]
           [ -out-dir=<output directory> ]
           [ -fw-ad ]
           [ -bi-ad-dir=<path to global declarations file> ]
           [ -default-inactive ]
           [ -default-full ]
           [ -default-full-ders ]
           [ -use-full-ders ]
           [ -use-sparse-ders ]
           [ -use-native-der-interface ]
           [ -compact-output=<compact strength> ]
           [ -wrap-output-at=<line number> ]
           [ -output-pass-dumps ]
           [ --help ]
```

The options `-base-file`, `-out-file` or `-out-dir`, and `-bi-ad-dir` are mandatory, others are optional. The option `-base-file` gives the name, and the absolute or relative path, to the input M-file to be processed for optimsation or AD. Similarly `-out-file` gives the name and path to the output file. Instead of an explicit output file, an output directory can be specified using `-out-dir`, where the output file created has the same name as the input with a `ad_` prefix added to it. MSAD also requires the path to the directory containing the meta-file `globals.prot` that provides the prototypes to builtins and global functions. This directory also contains the MAD package files to specialise. By default both are placed in the `MSAD/globals` directory. The `-bi-ad-dir` should therefore point to this directory. The `-default-inactive`, `-default-full`, `-default-full-ders`, `use-full-ders` and `-use-sparse-ders` options control the default lattice values used at the start, with obvious meanings in the context of AD. MSAD also plugs into the `bvp4cAD` solver that is familiar with the internal repre-

sentation of the AD derivatives. The option `-use-native-der-interface` can be used to indicate to MSAD to preserve the internal representation. The options `-compact-output` and `-wrap-output-at` control the formatting. The final option `-output-pass-dumps` controls the debugging output from MSAD. IR from various analysis and optimisation phases can be output to files. Please note some of these logs in textual form are verbose and can be very large.

Appendix F

Design Diagrams

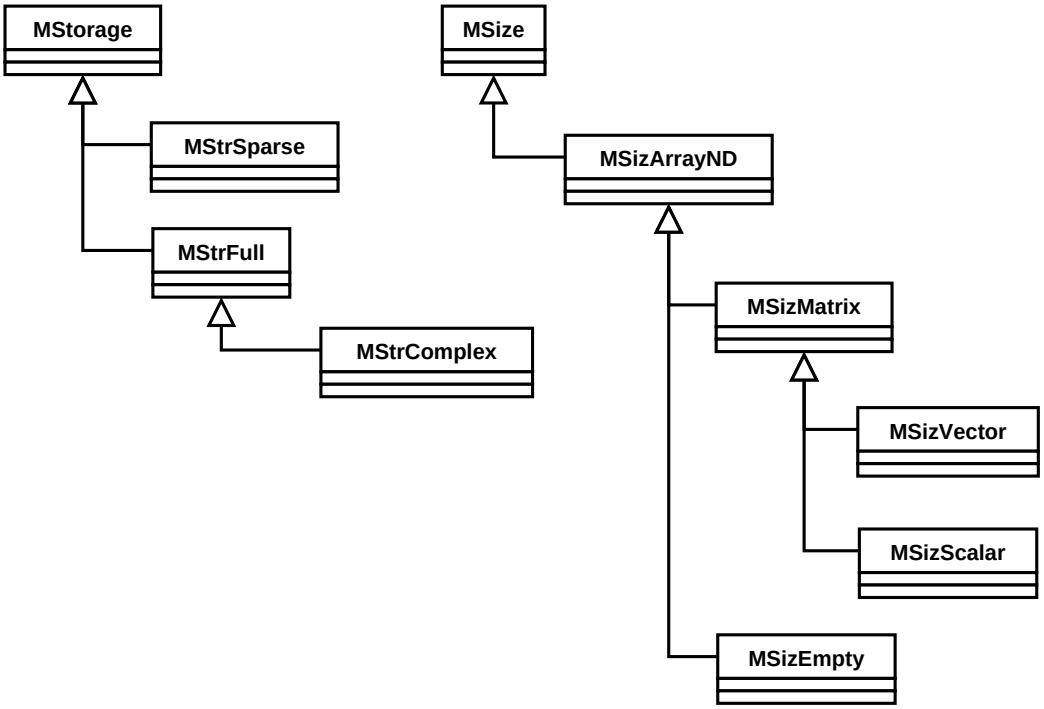

Figure F.1: MATLAB storage-type and array size – class hierarchy

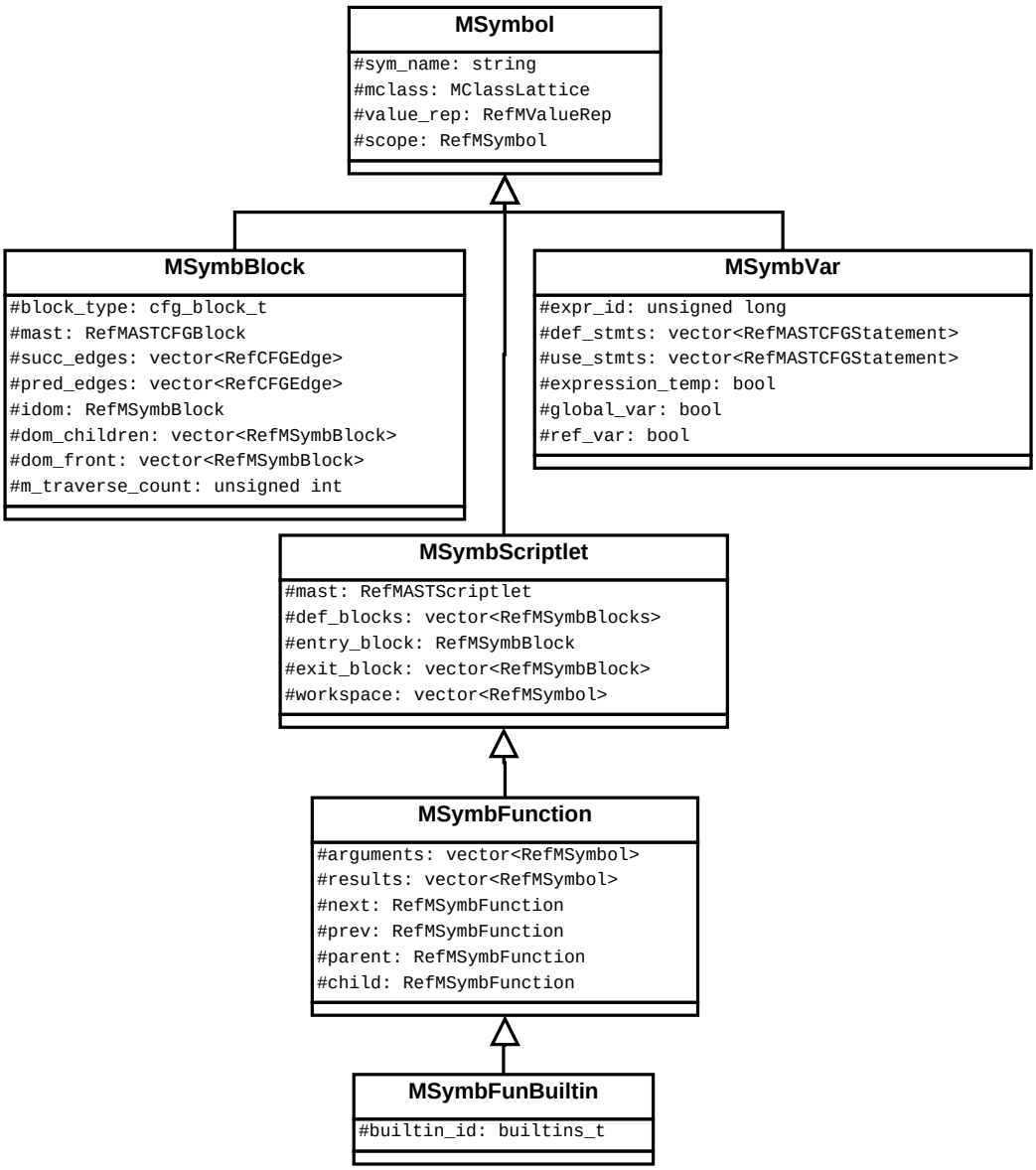

Figure F.2: MSAD symbol record - MSymbol class hierarchy

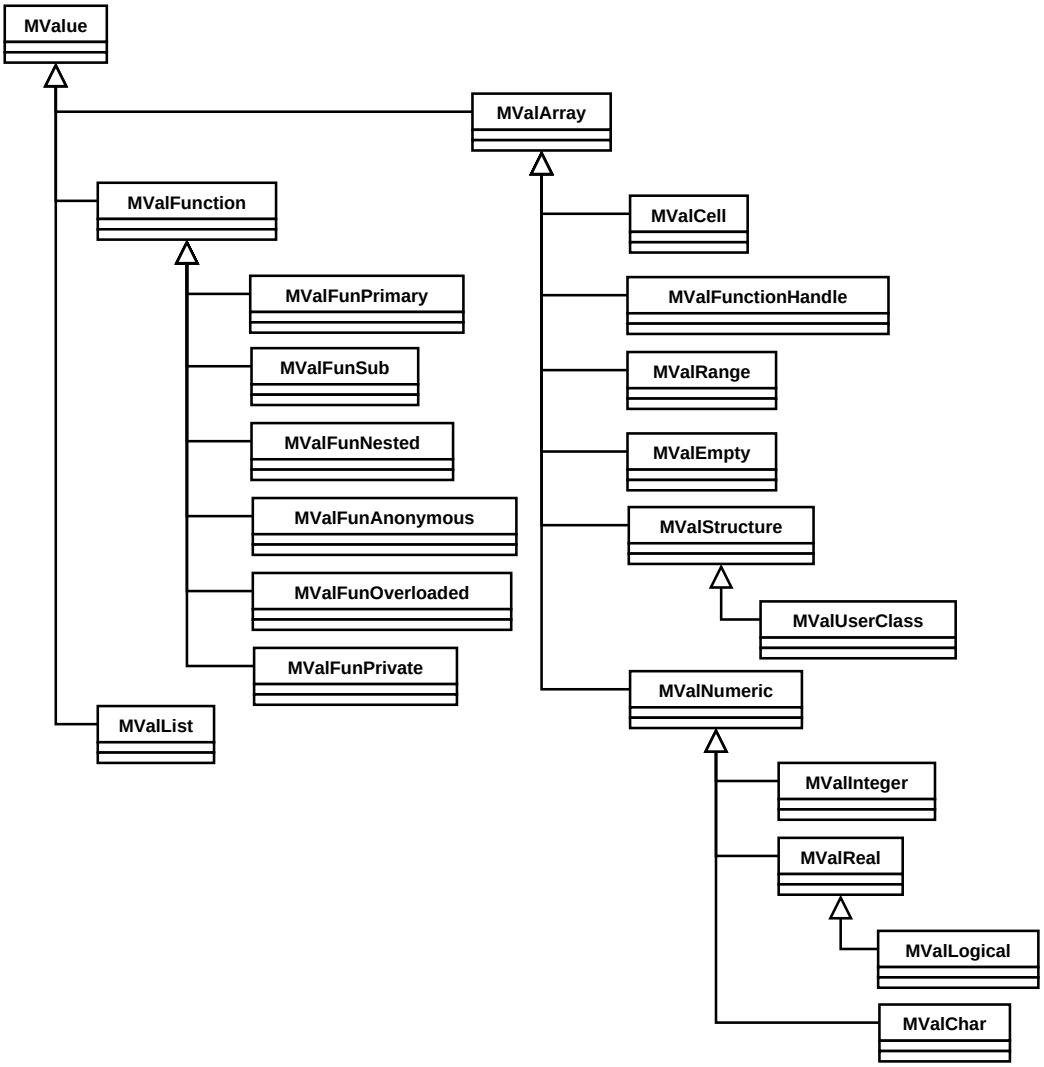

Figure F.3: MATLAB types – class hierarchy